Universitetet i Oslo
Institutt for informatikk

Databaseindekser i RAM, en ytelsesstudie

Cecilie Haaland Fritzvold

Masteroppgave

27. juli 2006

Forord

Denne masteroppgaven er utført ved forskningsgruppen "Objektorintering, modellering og språk" ved Institutt for Informatikk, Universitetet i Oslo. Oppgaven er veiledet av amanuensis Ragnar Normann, og bygger videre på to tidligere masteroppgaver av Tomas Are Haavet og Kjell-Magne Øierud. Kildekoden til Rindex er tilgjengelig på nettet:
`http://www.ifi.uio.no/forskning/grupper/oms/rindex/`

Takk til alle som har hjulpet meg underveis med masteroppgaven min. Takk til Ragnar Normann for meget god veiledning. Takk til Asbjørn Sannes for hjelp med å få tankene i gang igjen når jeg har sittet fast, og for hjelp med å finne godt skjulte feil i koden. Takk til alle venner og familie for god støtte.

Innhold

Kapittel 1

Innledning

Rindex står for RAM index, og er en plattform bygget for å brukes på toppen av eksisterende databasehåndteringssystemer (DBMS). Tradisjonelle DBMS'er er designet med tanke på at RAM er en begrenset ressurs. Dette er ikke lenger tilfelle. Dagens datamaskiner blir utstyrt med stadig større mengder RAM, og hensikten med Rindex er å utnytte dette til å forbedre ytelsen til et databasehåndteringssystem. Dette oppnås i Rindex ved å legge alle sekundærindekser i minnet, samtidig som dataene fremdeles lagres på disk. Rindex ble utviklet av Tomas Are Haavet og Kjell-Magne Øierud i løpet av deres hovedoppgave (se [3] og [12]), og denne oppgaven er et videre arbeid med Rindex-plattformen.

Det er visse begrensninger i Rindex slik plattformen opprinnelig er implementert, den viktigste er at Rindex ikke støtter oppdateringer. Målet med denne oppgaven har vært å utvide Rindex til å støtte oppdateringer. Dette har bydd på mange utfordringer, siden flere implementasjonsvalg i Rindex er gjort under antagelsen av at indeksene er statiske. Målet med plattformen etter utvidelsen er å oppnå bedre ytelse enn databasehåndteringssystemer som legger indekser på disk. Dette gjelder både for spørringer og oppdateringer.

Dette dokumentet har som formål å beskrive de utvidelsene som er gjort med Rindex-plattformen for å gi den støtte for oppdateringer, og å vise til ytelsestester som er gjort for å sammenligne Rindex med et databasehåndteringssystem som lagrer indekser på disk. Neste kapittel beskriver Rindex-plattformen før utvidelsene, og en analyse som ligger til grunn for valg av løsninger. De neste kapitlene beskriver det arbeidet som er gjort. Så følger to kapitler som beskriver de ytelsestestene som er gjort, og presenterer resultater av en sammenligning mellom ytelsen til Oracle brukt med og uten Rindex. Oracle er valgt både fordi det i utgangspunktet var Oracle som ble valgt som underliggende DBMS for Rindex, og fordi det er Oracle som er tilgjengelig ved instituttet.

Å gi Rindex støtte for oppdateringer betyr i praksis at Rindex må støtte

SQL's DDL[1] og DML[2]. Fordi det er DML-biten som er interessant i ytelsesøyemed, så er det her fokus er lagt. Kun helt grunnleggende `CREATE`- og `DROP`-setninger er støttet.

[1]Data Definition Language
[2]Data Manipulation Language

Kapittel 2

Beskrivelse og analyse av Rindex

2.1 Beskrivelse av Rindex 0.1.x

Rindex 0.1 er den første Rindex-versjonen som ble implementert. Etter arbeidet med den inverterte indeks-strukturen og substitusjonsoperatoren, beskrevet i [3], ble Rindex-versjonen som implementerer den inverterte indeksstrukturen med tilhørende substitusjonsoperator som viste seg å være best, døpt Rindex 0.1.1. Rindex 0.1.x er en fellesbetegnelse for Rindex 0.1 og Rindex 0.1.1.

2.1.1 Struktur og oppbygning

Rindex-plattformen er bygd opp av flere komponenter. Sentralt i Rindex er biblioteket librindex. Librindex består av ulike moduler som samarbeider med hverandre: Scanner, Parser, Analyzer, Optimizer, IndexManager, Executor, DBSchema og DBFrontend. Kommunikasjonen med librindex foregår ved hjelp av funksjoner definert der og de tre tegnstrømmene in, out og err.

Klassen DBSchema brukes for å holde på skjemainformasjonen til databasen. Denne bygges opp under oppstart av Rindex og inneholder informasjon om navn på relasjoner, og for hver relasjon: antall attributter og tupler, antall attributter i primærnøkkelen og navn og datatype til hvert attributt. For å kommunisere med DBMS'en brukes klassen DBFrontend som har funksjoner for å gjøre spørringer mot databasen og hente ut informasjon fra resultatsettet. DBFrontend er en abstrakt klasse som det kan lages subklasser av for å implementere støtte for flere forskjellige underliggende DBMS'er. Rindex har foreløpig støtte for Oracle og CSV-filer[1].

[1] "Comma Separated Values". Dette er ikke en DBMS, men fungerer som en datakilde.

For å kommunisere med Rindex er det laget en kommandotolker som fungerer som en klient som kommuniserer med en server. Kommandotolkeren kommuniserer med Rindex-serveren med en enkel stop and wait protokoll som benytter en pipe-fil. Dette innebærer at klienten sender kommandoer til Rindex, og venter på å få en bekreftelse tilbake før den sender en ny kommando. Kommandoene sendes via en pipe-fil, som både klienten og Rindex lytter til.

2.1.2 Indeksene

Indeksene i Rindex håndteres av klassen IndexManager. Denne klassen tar seg av alle operasjoner på indeksene. Dette gjør at det er enkelt å bytte ut denne modulen for å teste ut flere forskjellige måter å håndtere indeksene på.

Indeksene i Rindex lagres som sorterte sekvensielle filer hvor hver indekspost består av et par med attributtverdi og primærnøkkel. Fordi indeksene i Rindex er sekundære, fungerer primærnøkkelen her som en peker til lokasjonen attributtverdiens tilhørende tuppel befinner seg på i databasen. Rindex indekserer alle attributter som har en datatype som er støttet av plattformen. Dette gjelder heltall, flyttall og strenger på maks 256 tegn.

Rindex lager også inverterte indekser for effektivisering av oppslag. Disse benyttes blant annet til projeksjoner og substitusjoner.

Indeksene genereres ved oppstart og lagres permanent i minnet. Rindex må hente ut alle relasjonene med attributter det skal genereres indekser på fra databasen, og det er naturlig at dette tar noe tid.

Følgende indeksoperasjoner er implementert i Rindex:

Seleksjon En seleksjon kan kun ha en betingelse. For å støtte seleksjon med flere betingelser benyttes oppsplittingsregler for `AND` og `OR`. Seleksjoner med sammenligning av to attributter støttes ikke.

Union Dette er unionsoperatoren som benyttes for å gjøre en `OR`-operasjon. Det er en mengdeunion.

Equi-join Dette er en join der joinattributter med NULL-verdi ikke kommer med i resultatet. Den skiller seg fra en `NATURAL JOIN` ved at joinattributtene kan ha forskjellige navn, og dermed må disse spesifiseres i spørringen.

Projeksjon Projeksjonen tar et sett med attributter og en indeks, og generer et resultatsett som skrives til standard-ut.

Substitusjon Denne operasjonen bytter ut attributtet i en indeks med et attributt med samme primærnøkkel. Dette gjøres for å kunne gi rett input til andre operasjoner.

Indekssortering Indeksene sorteres på attributtverdiene og sorteringen foregår med quick-sort.

2.1.3 Eksekvering

Modulene Scanner, Parser og Analyzer utgjør tilsammen en kompilator som skal sjekke at en spørring er syntaktisk korrekt, bygge et syntakstre og sjekke semantikken til spørringen. Parseren godkjenner enkle `SELECT`-setninger med `FROM`-, `INNER JOIN`-, `WHERE` og/eller `ORDER BY`-ledd. Aritmetiske operasjoner på attributter, seleksjon på to attributter og join med flere betingelser er ikke støttet i denne versjonen av Rindex. Hvis syntaksen godkjennes av parseren, startes den semantiske sjekken. Dette gjøres ved en postfiks traversering av syntakstreet. Hvis en syntaktisk eller semantisk feil oppstår et eller annet sted under sjekkingen, avsluttes sjekkingen umiddelbart, og det gis en feilmelding.

Etter returnering av syntakstreet sendes dette til Optimizer-klassen som har som oppgave å oversette dette til et spørretre, optimalisere spørretreet og generere en eksekveringsplan fra det optimaliserte spørretreet. Det er kun noen få optimaliseringer som er implementert i Rindex, men det er lagt til rette for å utvide med flere. Rindex genererer ikke flere mulige eksekveringsplaner slik som er vanlig, men nøyer seg med en.

Til slutt sendes eksekveringsplanen til Executor som utfører selve eksekveringen og returnerer resultatsettet.

2.1.4 Begrensninger

Rindex-plattformen, slik den er i dag, har endel mangler og begrensninger. Det er ikke støtte for relasjoner med attributter som ikke blir indeksert, det gjøres ingen forsøk på optimaliseringer av spørreplanen, det er kun et lite subsett av SQL-spørringer som støttes, og NULL-verdier støttes ikke. Det er mange flere forbedringer som kan gjøres, men den viktigste begrensningen på plattformen er mangelen på støtte av oppdateringer. Rindex er bygget på en antagelse om at indeksene er statiske, noe som ikke gjelder dersom oppdateringer skal støttes. Indeksstrukturen i Rindex, samt operasjonene på disse, må revurderes hvis Rindex skal støtte oppdateringer. Det er dette vi skal se nærmere på her.

2.2 Problemstillinger

Versjon 0.1.x av Rindex ble laget under forutsetning av at dataene i databasen er statiske. Dette innebærer i hovedsak to ting: Endringer på dataene støttes ikke i noen form og koden er skrevet på en slik måte at det ikke bare er å legge til støtte for oppdateringer på dataene uten å endre på valg av datastrukturer. Når noe skal gjøres med dette, er det to problemstillinger

datastrukturene byr på. Den første er om det overordnede valget av datastruktur er hensiktsmessig med tanke på at kravene til en datastruktur som skal støtte oppdateringer er helt annerledes enn kravene til en datastruktur for lagring av statiske data. Hovedproblemet er at indeks-strukturen til Rindex benytter sorterte data lagret sekvensielt. Det er et kjent faktum at en slik datastruktur ikke er egnet når det kan forekomme endringer i dataene. Dette problemet skal vi komme tilbake til senere. Den andre problemstillingen er mer en programmeringsteknisk detalj. Er datastrukturene implementert på en slik måte at det i det hele tatt er mulig å gjøre endringer på dataene? Hovedproblemet her ligger i bruken av arrayer. Disse får allokert en gitt plass i minnet ved opprettelse, og hvis man senere skal legge til nye elementer, vil det ikke være plass til disse uten å gjøre en reallokering.

2.3 Oppdateringer med eksisterende datastruktur

Det er i hovedsak to deler av Rindex som er berørt av antagalsen om statiske data: implementasjonen av indeksene (seqfile_indexes.cpp) og lagringen av metadata om relasjonene (dbschema.hpp/dbschema.cpp). Alt av relasjoner, attributter, primærnøkler, indekser, attributtverdier og lignende lagres sekvensielt i minnet ved bruk av arrayer som det allokeres plass til én gang. Ingen av disse arrayene er utvidbare i den forstand at man kan legge til elementer. Rindex er programmert i C++, og C++ har noe som kalles STL[2]. Her finner man blant annet en vektor, som er en utvidbar array som kan aksesseres på samme måte som en vanlig array med `[]` operatoren. Dette er et alternativ til de vanlige arrayene. En implementasjon som gjør bruk av vektorer er beskrevet i kapittel 4.

2.4 Alternativet: en ny datastruktur

Fordi det første forsøket på å utvide Rindex til å støtte oppdateringer ikke nødvendigvis er det mest hensiktsmessige med tanke på det opprinnelige valget av datastrukturer (sekvensiell lagring), ønsket jeg å se på mulige forbedringer av selve datastrukturen. Da det sjelden er så mange relasjoner og attributter i en database at sekvensiell lagring av disse er et problem, er det kun en del av Rindex som er interessant å se på i denne forbindelsen, og det er indeksene (seqfile_indexes.cpp). Valget falt på CSB^+-trær, som er en variant av B^+-trær som er optimalisert for å ligge i minnet og utnytte cachen best mulige. CSB^+-trær er beskrevet i [10], og min implementasjon er beskrevet i kapittel 5. Den nye indeksstrukturen, basert på bruken av CSB^+-trær, er beskrevet i kapittel 6.

[2]Standard Template Library

Kapittel 3

Statisk analyse og plangenerering

Før eksekvering av en spørring utfører Rindex flere typer analyser og omgjøringer som til slutt skal lede fram til en eksekverbar plan. Som nevnt i 2.1.3, blir spørringen først sendt gjennom en skanner som deler den opp i såkalte tokens. Deretter blir spørringen sjekket for syntaktisk korrekthet av en parser som leser tokens fra skanneren. Parseren genererer et syntakstre som så må analyseres for semantisk korrekthet. Hvis spørringen både er syntaktisk og semantisk korrekt, blir det generert et spørretre utfra syntakstreet. Dette spørretreet blir så optimalisert og en eksekverbar plan generert. Alle disse delene er beskrevet i [3]. Når et større subsett av SQL skal implementeres, og alle spørringer ikke lenger begynner med `SELECT`, så berører dette alle disse stegene, og spesielt strukturen på alle trærne som er benyttet: syntakstreet, spørretreet og plantreet. Endringene og utvidelsene som er blitt gjort beskrives i dette avsnittet.

3.1 Skanneren

Skanneren til Rindex genereres av verktøyet flex (se [8]). Dette gjør det enkelt å gjøre utvidelser i form av å la skanneren gjenkjenne flere leksemer som blir tilordnet tokens. Nye leksemer som er lagt til er blant andre "`CREATE TABLE`", "`INSERT INTO`" og "`DELETE FROM`".

3.2 Parseren

Rindex' opprinnelige parser, laget med Bison (se [2]), støttet kun et subsett av SQL's `SELECT`-setninger som beskrevet i [3]. For å kunne tillate oppdateringer i Rindex må parseren akseptere oppdaterings-setninger. Parseren måtte derfor utvides til å støtte et subsett av SQL's DDL og DML. For å få

fram den grunnleggende funskjonaliteten oppdateringer krever, uten å måtte legge vekt på tunge detaljer en fullstendig implementasjon i henhold til SQL-standarden ville kreve, er kun støtte for enkle `CREATE`-, `DROP`-, `INSERT`-, `UPDATE`- og `DELETE`-setninger implementert. `ALTER`-setninger er blitt utelatt av denne grunn. Disse ville ikke bringe fram interessante nok endringer i denne sammenhengen, og det er bedre å legge opp til at det skal være enkelt å implementere dette senere. Den utvidete grammatikken er beskrevet i appendix A.

Når det gjelder syntakstreet som genereres av parseren, er dette nå utvidet med flere typer noder slik at strukturen i `CREATE`-, `DROP`-, `INSERT`-, `UPDATE`- og `DELETE`-setninger også kan representeres.

3.3 Semantikksjekker

De nye nodene i syntakstreet må også definere check_semantic() funksjonen slik at semantikken til de nye typene spørringer kan sjekkes på samme måte som for `SELECT`-setningene. Følgende sjekkes for oppdaterings-setningene:

- `CREATE`
  - Relasjonen eksisterer ikke fra før.
  - Det er spesifisert en primærnøkkel. Flere attributter kan være spesifisert som primærnøkkel. De vil da utgjøre deler av samme primærnøkkel.
- `DROP`
  - Relasjonen eksisterer.
- `INSERT`
  - Relasjonen eksisterer.
  - Det er spesifisert like mange verdier som attributter. Hvis ingen attributter er spesifisert, må det være oppgitt like mange verdier som relasjonen har attributter.
  - Hvis ingen attributter en spesifisert, må listen over attributter ekspanderes til alle attributter i relasjonen.
  - Hvis det er spesifisert attributter, må alle disse eksistere i relasjonen.
  - Typene til verdiene må matche typene til attributtene de skal settes inn i.
- `UPDATE`
  - Relasjonen eksisterer.

  - Alle attributtene er del av relasjonen.
  - Verdiene har typer som matcher attributtene de skal tilordnes.
  - Hvis det er et `WHERE`-ledd, sjekkes denne på samme måte som for `SELECT`-setninger.

- `DELETE`
  - Relasjonen eksisterer.
  - Hvis det er et `WHERE`-ledd, sjekkes denne på samme måte som for `SELECT`-setninger.

3.4 Spørretre

Med unntak av `UPDATE`- og `DELETE`-setninger som kan inkludere et `WHERE`-ledd som må eksekveres, så har et spørretre liten hensikt for oppdateringssetninger. All den nødvendige informasjonen for å kunne opprette en ny relasjon, slette en relasjon eller legge til en rad i en relasjon er tilgjengelig i syntakstreet. Det genereres derfor ikke noe spørretre for `CREATE`-, `DROP`- eller `INSERT`-setninger. `UPDATE`- og `DELETE`-setninger derimot trenger et spørretre for å få eksekvert `WHERE`-leddet slik at man vet hvilke tupler som berøres. Roten i et spørretre i Rindex er en projeksjonsnode. I `UPDATE`- og `DELETE`-spørringer foregår det ingen projeksjon på samme måte som i `SELECT`-spørringer. Det er likevel nødvendig å ha en liste over attributter som skal endres for `UPDATE`-spørringer. For å kunne integrere best mulig med den allerede eksisterende koden for optimalisering og plangenering viste det seg hensiktsmessig å lage to subklasser av projeksjonsnoden som vil bli røtter i spørretrærne til `UPDATE`- og `DELETE`-setningene. Disse får da den samme rollen som den opprinnelige projeksjonsnoden, men varianten for `UPDATE` lagrer en liste med sett av attributter og deres nye verdi, mens `DELETE` varianten ikke behøver å lagre noen slik liste i det hele tatt, siden alle attributter i en rad blir slettet.

3.5 Optimalisering og plangenering

Av samme grunn som over er dette kun nødvendig for `UPDATE`- og `DELETE`-setningene. I tillegg håndteres det på samme måte som for `SELECT`-setninger, så her er det få endringer. Det eneste som er nødvendig er at nodene i spørretreet til `UPDATE`- og `DELETE`-setninger også implementerer make_plan() funksjonen slik at en plan kan bli generert.

Kapittel 4

Sekvensielle indekser

Jeg ønsket i første omgang å gjøre et forsøk på å implementere støtte for oppdateringer i Rindex uten å måtte endre på noen valg av datastrukturer. Det er flere grunner til dette, blant annet at jeg ønsket å se om det overhodet var mulig. Dette var likevel ikke hovedmotivasjonen for å gjøre det. Jeg har brukt dette første forsøket på å skaffe meg innsikt i hvordan Rindex er implementert. For det første har dette gitt meg en mulighet til å bli fortrolig med koden til Rindex, som er ganske stor. Likevel har det viktigste ved denne implementasjonen vært å identifisere alle steder i koden som er påvirket av at Rindex kun støtter statiske data. Det er viktig å få identifisert disse slik at det er mulig å gjøre vurderinger om hva som må redesignes, og hva som kan gjenbrukes, muligens i noe modifisert form. En annen fordel med å først gjøre en naiv og "rett fram"-implementasjon som beholder sorterte sekvensielt lagrede data, er å ha et sammenligningsgrunnlag for en implementasjon som bruker en datastruktur som på papiret er mer egnet til oppdateringer.

Dette kapittelet beskriver kun oppdateringer av indeksene. Oppdateringer av dataene på disk er beskrevet i kapittel 7.

4.1 Generelle utfordringer med datastrukturen

Som nevnt tidligere, er det i hovedsak to deler av Rindex som er påvirket av bruken av kun statiske data: indeksene og metadataene. I tillegg er det andre, mindre problemer rundt om i koden som i stor grad er basert på at Rindex kun støtter `SELECT`-setninger. Dette er mest synlig i den delen av koden som behandler kompilering av spørringer og kall på eksekvering av disse. Når det gjelder metadataene som lagres i DBSchema (denne er beskrevet i [3]), er problemet at relasjoner, attributter og primærnøkler lagres i ikke-utvidbare arrayer. Dette kan løses ved å bruke STLs vektor[5], som er utvidbar, samtidig som den tillater vanlige array-operasjoner. Selv om en vektor ved første øyekast kan virke som en perfekt erstatning for arrayer, så har den et par egenskaper som gjør den mindre effektiv og som gjør at

man må være forsiktig i bruken. En av disse egenskapene er det som gjør at vektoren kan opprettholde sekvensiell lagring i minnet selv når den vokser. Det allokeres nemlig ny plass i minnet hver gang den vokser, og alle elementer kopieres over til det nye stedet. Det kan derfor være lurt å først fortelle vektoren hvor stor den behøver å være før en rekke med elementer settes inn. Vektoren overlaster også noen operatorer (slik som =), som gjør at oppførselen av disse kan være noe annerledes enn man er vant med fra arrayer. En vektor er derfor ikke nødvendigvis mest egnet til bruk i et system som Rindex, i hvertfall ikke i slik utstrakt bruk som i denne implementasjonen. Et mer effektivt alternativ kunne være å lage sin egen implementasjon av utvidbare arrayer med egenskaper mer tilpasset behovet. Jeg fant dessverre ikke tid til å gjøre dette i denne omgang, og har derfor valgt vektorer som et kompromiss.

Når det gjelder lagringen av verdier i Rindex, så var dette gjort på en slik måte at det ikke var hensiktsmessig å bruke en vektor. Det ble nemlig allokert et sammenhengende område i minnet til lagring av alle verdier, i form av en array. Deretter ble det lagret pekere inn i dette minneområdet alle steder som skulle kunne aksessere verdiene. Det er flere problemer med bruken av vektorer til å lagre verdiene. Det ene er som nevnt over, mangelen på effektivitet. Som kjent er det verdier det er mest av i en database, så effektivitetsproblemet ville komme best til syne her. Et annet problem av mer praktisk art er at pekere inn i en vektor blir ugyldige ved innsetting på grunn av flyttingen av elementer i minnet. Det ville dermed ikke lenger være mulig å lagre pekere til verdiene. I steden måtte det vært lagret indekser inn i vektoren, men dette ville kreve at en stor del kode måtte bli skrevet om. Fordi tabellen som lagrer verdiene aldri aksesseres direkte (bortsett fra ved initialisering), men kun via pekere, er det et annet alternativ for lagring av verdiene som virker mer hensiktsmessig. Dette alternativet er blitt implementert, til tross for at også dette har svakheter. Hver gang en ny verdi skal lagres, allokeres det plass til denne, og pekeren til elementet lagres overalt der det er nødvendig. Dette gjør at man aldri behøver å tenke på å utvide plassen i minnet. Ulempen med denne varianten er at gjentatte innsettinger og slettinger kan føre til fragmentering av minnet, noe som vil gå utover ytelsen.

4.2 Implementasjon av DDL

4.2.1 Create

Implementasjonen av støtte for `CREATE`-setninger innebærer å kunne gjøre endringer på metadataene i tillegg til å initiere indeks-strukturen slik at den kan lagre indekser til en ny relasjon. Endringer på metadataene vil i dette tilfellet si å kunne legge til en ny relasjon og nye attributter. Etter å ha gjort om de statiske arrayene som lagrer relasjonene og attributtene

til STL-vektorer, er det rett fram å kunne legge til nye. Relasjons-id'er og attributt-id'er genereres etterhvert som de trengs. En ny id vil alltid være neste tall i rekken etter den største som er i bruk. Id'er som ikke lenger er i bruk etter at en relasjon er blitt slettet, vil ikke bli gjenbrukt. Etter at metadataene er endret, blir det opprettet en ny indeks for hvert attributt i den nye relasjonen. Disse vil da kunne fylles opp med verdier etterhvert.

4.2.2 Drop

Å slette en hel relasjon må gjøres i to steg. Først må alle indeksene relatert til denne relasjonen slettes, og deretter kan relasjonen fjernes fra metadataene. Alt minnet som er blitt allokert til informasjon om denne relasjonen frigjøres. Unntaket er plasser i tabeller som er indeksert med attributt-id'er eller relasjons-id'er. Det er endel slike tabeller både i indeks-strukturen og i metadataene, og de inneholder pekere til objekter eller andre tabeller relatert til relasjonen eller attributtet. Etter å ha frigjort minnet disse pekerne peker til, må derfor disse plassene i tabellene settes til NULL. Dette påvirker endel andre funksjoner, som løper igjennom slike tabeller fra 1 til n uten å ta hensyn til at alle id'er ikke nødvendigvis er i bruk. Dette var jo heller ikke nødvendig før man kunne slette relasjoner. Det ble derfor nødvendig under implementasjonen av drop å gå gjennom alle disse funksjonene og legge inn tester for om pekere er satt til NULL, og da hoppe over disse.

4.3 Implementasjon av DML

4.3.1 Insert

Ved insert skal det for hvert attributt i relasjonen det skal settes inn i, legges til en ny verdi. Fordi en `INSERT`-setning ikke nødvendigvis spesifiserer verdier for alle attributtene, må det legges til default-verdier for de attributtene som ikke en spesifisert. Her er det greit å la 0 være default-verdi for alle tall, og den tomme strengen være default verdi for alle tegn-baserte typer. Foreløpig er det ikke støttet å kunne spesifisere en default-verdi for hvert attributt ved `CREATE`, men dette kan enkelt legges til. Default-verdiene til uspesifiserte attributter i `INSERT`-setningene vil da hentes i metadataene. Først gjøres all initialiseringen som er nødvendig ved innsetting av en ny rad med default-verdier for alle attributtene. Deretter endres verdiene der attributtene har fått spesifisert en verdi i `INSERT`-setningen. Dette gjør det enkelt å opprette en ny rad, uavhengig om det er spesifisert verdier for alle attributter eller ikke. Innsettingen skjer i utgangspunktet ved å legge de nye verdiene til på slutten. Da er ikke lenger kravet om at verdiene skal være sortert, oppfylt. En må derfor gjøre en sortering for å ordne på dette. Siden vi vet at alle verdier utenom den siste raden allerede er sortert, er det nok å finne den riktige plassen i lista for så å skyve alle verdier derfra en plass nedover. Da kan den

nye verdien flyttes opp til riktig sted. Innsetting får da tids-kompleksitet O(n) der n er antall rader i relasjonen det settes inn i.

4.3.2 Update

Ved update må man først finne ut hvilke tupler som skal få endrede verdier. Dette gjøres på samme måte som for spørringer, og en kan dermed bruke de samme funksjonene for å finne de riktige tuplene som brukes for spørringer. Etter dette kan de attributtene som er spesifisert i `UPDATE`-setningen få endret sine verdier i de riktige radene. Etter at denne endringen er gjort, har vi samme problem som ved innsetting, nemlig at dataene ikke lenger er sortert. En tilsvarende sortering kan gjøres også her, med den forskjell at verdiene i lista kan bli dyttet oppover i steden for nedover hvis den nye verdien er mindre enn den gamle. Problemet med denne strategien ved update er hvis det er mange rader som skal oppdateres. Tidskompleksiteten er nemlig $O(n * m)$ der n er antall rader i relasjonen, og m er antall rader som skal oppdateres. En annen mulighet er å først oppdatere alle verdiene, for så å bruke quicksort til å sortere dem igjen. Da blir tidskompleksiteten $O(n * log(n))$. Problemet med denne varianten er at quicksort er kjent for å være dårlig når dataene er delvis presorterte. Dette er likevel ikke et problem hvis det er valgt en god pivot. En mulig løsning er å velge sorteringsmetode ut fra hvor stor m er. Hvis m er liten brukes den første varianten til å sortere, men hvis m er stor brukes quicksort. Kun sortering med quicksort etter at alle verdier er oppdatert, er blitt implementert. Dette er fordi quicksort er rask uansett, noe som gjør at det vil være en grei løsning i alle tilfeller. Det var i tillegg enklest å gjøre det slik.

4.3.3 Delete

Delete viste seg å være vanskelig å implementere med denne indeks-strukturen. Det er nemlig ikke nok å bare finne fram til alle verdiene som berøres, slik som for update, man må finne igjen alle steder som lagrer pekere til verdier, eller til objekter som skal slettes. Indeks-strukturen er laget slik at det skal være enkelt å få tak i de objektene og dataene man trenger via pekere, men den er derimot ikke laget for å kunne finne igjen alle steder som peker til et gitt objekt. Dette gjør det vanskelig å få slettet de riktige radene. Det ville vært nødvendig med sekvensiell søking gjennom to forskjellige tabeller, i tillegg til en for hvert attributt i relasjonen. Alle disse tabellene har like mange rader som relasjonen, og de kan ikke søkes igjennom i parallell, da de er sortert forskjellig. Selve slettingen ville også medføre en flytting av verdier oppover eller nedover for å tette hullene. Tidskompleksiteten ville altså vært $O(2 * n * m * a) = O(n * m * a)$ der n er antall rader i relasjonen, m er antall rader som skal slettes og a er antall attributter. Delete er derfor ikke blitt implementert i denne versjonen, selv om det hadde vært mulig.

Kapittel 5

Cache Sensitiv B^+-tre implementasjon

Forrige kapittel beskrev en implementasjon av støtte for oppdateringer i Rindex på den eksisterende datastrukturen. Vi så at det var endel utfordringer forbundet med å gjøre dette på en sekvensiell datastruktur. Dette kapitlet beskriver implementasjonen av en annen mulig datastruktur for indeksene, CSB^+-trær. Neste kapittel beskriver en implementasjon av støtte for oppdateringer basert på denne datastrukturen.

[10] beskriver et alternativ til B^+-trær som er utviklet med tanke på lagring av data i minnet i steden for på disk. Ut fra resultatene presentert i denne artikkelen har jeg utviklet et Cache Sensitivt B^+-tre (CSB^+-tre) til bruk i Rindex. Grunnprinsippene for datastrukturen og algoritmene på denne er hentet fra [10], men alt er implementert fra bunnen av og tilpasset behovene til Rindex.

5.1 Valg av CSB^+-tre

[10] beskriver tre varianter av CSB^+-trær som alle viser seg å være overlegne i forhold til B^+-trær når det gjelder søketid på data som ligger i minnet. Når det gjelder innsetting, er derimot to av de tre variantene av CSB^+-trær tregere enn B^+-trær. De to variantene som er tregere på innsetting kalles vanlige CSB^+-trær og segmenterte CSB^+-trær. Grunnen til at disse er tregere ved innsetting er måten de håndterer CSB^+-træres styrke: bruken av *nodegrupper*[1]. Ved splitting av en node allokeres det ny plass i minnet til en større nodegruppe, og alle nodene fra den gamle nodegruppa kopieres over i den nye. Den tredje varianten, fulle CSB^+-trær bøter på dette problemet ved å allokere nodegrupper av maksimal størrelse med en gang. Det vil si at hvis en intern node har plass til n nøkler, så kan den maks ha n+1 barn,

[1] En nodegruppe er en gruppe av noder som er lagret sekvensielt i minnet

og nodegruppen som skal romme disse barna må derfor ha en størrelse på $(n+1)*$ nodestørrelsen. Ved å allokere plass til nodegrupper av full størrelse med en gang, behøver ikke noder å kopieres fra ett sted til et annet i minnet ved splitting. Ulempen med denne varianten er at den krever mer plass enn de andre variantene, siden den tar opp ubrukt plass i minnet. Til gjengjeld er den like rask eller raskere enn de andre variantene på alle operasjoner. Siden forutsetningen for Rindex er at vi har mer enn nok minne tilgjengelig, anser jeg ikke minnebruken for et problem, og valget falt derfor på fulle CSB^+-trær.

5.2 Datastrukturen

Fordelen med et CSB^+-tre er at ved å samle noder i nodegrupper der nodene ligger etter hverandre i minnet, trenger man kun å lagre en barn-peker i de interne nodene. Denne pekeren kan brukes til å aksessere alle nodene i nodegruppen ved å benytte et offsett. Dette gjør at man kan lagre dobbelt så mange nøkler i en intern node (forutsatt at en nøkkel og en peker tar like mye plass), uten at det øker størrelsen til noden. Det samme prinsippet kan også brukes på bladnodene der det er vanlig i et B^+-tre å lagre søskenpekere. Siden alle noder i en nodegruppe kan aksesseres ved å legge til eller trekke fra et offsett på minneadressen, er det nok å lagre søskenpekere til nabonoder i en annen nodegruppe. Altså trenger kun den første og siste noden i en nodegruppe å lagre søskenpekere. For å slippe å skille på bladnodene settes det av plass til en søskenpeker i alle bladnoder uavhengig av om de er på enden av en nodegruppe eller ikke.

De interne nodene har følgende innhold:

- Antall nøkler
- Barn-peker
- Liste av nøkler

Bladnodene lagrer følgende elementer:

- Antall nøkkel/data-par
- Søskenpeker
- Liste med nøkler
- Liste med data

Et dataelement i en bladnode er en peker til det stedet i minnet der det nøkkelen skal identifisere er lagret. Dette må håndteres utenfor treets datastruktur.

Siden hensikten med et CSB$^+$-tre er å utnytte minnet og cachen best mulig, er det naturlig å se på hvordan cachen er organisert. Cachen er delt inn i blokker, kalt cache lines. Når prosessoren må lese fra minnet og inn i cache, eller skrive fra cachen tilbake til minnet, overføres en cache line. Hver cache line har sin egen indeks, i tillegg til en tag som referer til minneadressen i hovedminnet (RAM) ([1], [11]). For å utnytte cachen best mulig bør derfor en node ha samme størrelse som prosessorens cache line. Da kan en hel node overføres om gangen, og vi slipper unna med én leseoperasjon mellom minnet og cache for hver node som skal aksesseres. Fordi cache line størrelsen kan være forskjellig fra prosessor til prosessor, har jeg laget et lite assembler-program som finner denne størrelsen. Dette programmet kan da kjøres "compile-time" og resultatet bli gitt til kompilatoren ved hjelp av en DEFINE. På testmaskinene til rådighet for dette prosjektet er cache linen på 64 byte, som er den mest vanlige størrelsen på dagens prosessorer. Andre prosessorer kan ha en cache line størrelse på f.eks 32 byte eller 128 byte. De følgende utregningene er basert på en cache line på 64 byte.

For å få en node til å ha samme størrelse som cache linen er det antall nøkler som må tilpasses. Hvis antall nøkler lagres som en int (som tar opp 4 byte), og en peker tar opp 4 byte, vil det si at vi har 56 byte igjen til rådighet til nøkler i de interne nodene og til nøkkel/data-par i bladnodene. Siden et data-element er en peker, tar også denne opp 4 byte, så hvor mange nøkler og par vi kan lagre avhenger av nøkkel-størrelsen. En nøkkel må kunne være alle typer data-elementer Rindex indekserer, altså 4 byte heltall, 8 byte heltall, 4 byte flyttall, 8 byte flyttall og tekst. Tekst lagres som en char-peker i vanlig C-stil og tar opp 4 byte. Med 4 byte nøkler vil vi kunne lagre 14 nøkler i de interne nodene og 7 nøkkel/data-par i bladnodene. Med 8 byte nøkler blir det kun plass til 7 nøkler i de interne nodene og 4 nøkkel/data-par i bladnodene. Her oppstår det også et problem siden 4 nøkkel/data-par der nøklene tar opp 8 byte mens data-pekeren tar opp 4 byte tilsammen utgjør 48 byte, og ikke 56 byte. Disse bladnodene må derfor paddes slik at de fyller nøyaktig 64 byte.

For å slippe å implementere forskjellige noder for alle nøkkeltypene er nodene implementert som C++-templates. Antallet nøkler, nøkkel/data-par og hvor mye som må paddes i bladnodene, beregnes ved hjelp av makroer som tar nøkkel-typen som parameter og bruker cache line størrelsen som grunnlag. Et problem som oppstår fordi barn-pekeren i en intern node både kan peke til en annen intern node og til en bladnode, er at det må være mulig å lagre de to typene pekere som samme type peker, samtidig som man må kunne finne tilbake til den riktige typen objekt. C++ tilbyr en mulighet til å gjøre dette ved å la de to klassene arve fra en en tredje klasse. Hvis denne tredje klassen er virtuell, vil man kunne benytte dynamic_cast til å finne ut hvilken subklasse pekeren opprinnelig peker til. Problemet med denne løsningen er at en virtuell klasse automatisk opptar 4B, selv om den ikke lagrer noen elementer. Selv om det elementet som er felles for de to nodetypene (antall

nøkler og antall nøkkel/data-par) kunne vært lagret i den virtuelle klassen for å fjerne disse ekstra 4B (de legges til fordi den virtuelle klassen ikke kan være tom), er dette fremdeles ikke en akseptabel løsning fordi det oppstår tilgangsproblemer til arvede attributter ved bruk av templates. Løsningen som er benyttet baserer seg på at antallet nøkler aldri vil overskride 255 (det største tallet som kan lagres i én byte). Ved å endre attributtet som lagrer antall nøkler (eller antall nøkkel/data-par) fra en int til en char-array på 4B får man plass til fire elementer av én byte. Antallet nøkler lagres da i den første byten, og et tall som identifiserer om en node er en bladnode eller en intern node lagres i den andre byten. I tillegg er det to uutnyttede bytes som kan brukes til å lagre annen informasjon hvis det skulle bli nødvendig. En funksjon is_leaf() som er implementert i begge typene noder, sjekker dette tallet og returnerer true hvis noden er en bladnode og false ellers. Denne løsningen gjør at man ikke behøver å redusere antall nøkler og nøkkel/data-par for å kunne skille de to node-typene fra hverandre.

Selve funksjonaliteten til treet ligger i en klasse kalt CSB_Tree. Denne er implementert som en template som tar to parametre, nøkkeltypen og en sammenligningsfunksjon som skal returnere true hvis det første elementet er minst. Klassen lagrer en peker til rotnoden i treet i tillegg til en peker til første element i en liste over alle nodegruppene. Denne listen brukes når hele treet skal slettes for å enkelt kunne frigjøre alt minnet som er allokert. I tillegg implementerer CSB_Tree en iterator til bruk ved søk i treet. Et søk vil returnere en peker til en iterator som så kan brukes til å hente ut alle elementer som er ønsket (f.eks alle elementer som er større enn det det ble søkt på). Iteratoren har enkle operasjoner som å gå til neste eller forrige element og å returnere nøkkelen og data-elementet iteratoren peker på.

5.3 Operasjonene

Her følger en beskrivelse av de operasjonene som kan utføres på treet.

5.3.1 Bulkload

Denne funksjonen benyttes til å laste en sekvens av elementer inn i et initielt tomt tre. Innsettingsmetoden som er benyttet i bulkload er mer effektiv enn å utføre en sekvens av insert (dette vises både av resultater i [10] og av egne testresultater). Parametre til funskjonen er antall elementer som skal settes inn, en array med nøkler som må være sortert i stigende rekkefølge, og en array med data-elementer som hører til nøklene. Treet bygges opp fra bunnen ved å allokere nok nodegrupper og sette inn elementene. Deretter beregnes og opprettes det, nivå for nivå, interne noder, helt til det kun er behov for en intern node, som blir rot i treet. Denne metoden har både fordeler og ulemper. Den store fordelen er at det går mye fortere siden man slipper både å søke seg nedover i treet for hver innsetting, og å splitte noder.

En opplagt ulempe er at hvis dataene ikke i utgangspunktet er sortert på nøklene, så må de sorteres før de kan settes inn. For å undersøke om dette ville føre til at bulkload likevel ikke var raskere enn å utføre en sekvens av insert, dersom tiden for sortering er regnet med, er det utført noen tester. For å oppnå signifikantene resultater genereres en sekvens på 10 millioner tilfeldige tall. Disse sorteres ved hjelp av quicksort, og settes så inn med bulkload. Tiden kan så sammenlignes med tiden det tar å utføre 10 millioner insert operasjoner med de samme dataene. Testene er utført uavhengig av hverandre, det eneste som er felles er dataene. De følgende resultatene er basert på snittet av 100 kjøringer på en dedikert maskin. Testmaskinen har en AMD Athlon 64 3500 prosessor som kjører på 2.2GHz og har 512KB cache. Cache linen er som nevnt på 64 bytes. I tillegg har maskinen 1GB RAM. En graf fra forskjellige kjøringer er vist i figur 5.1 på neste side.

```
Bulkload of  10000000  elements:
Quicksort: 2.525
Bulkload: 0.310
Total time: 2.836

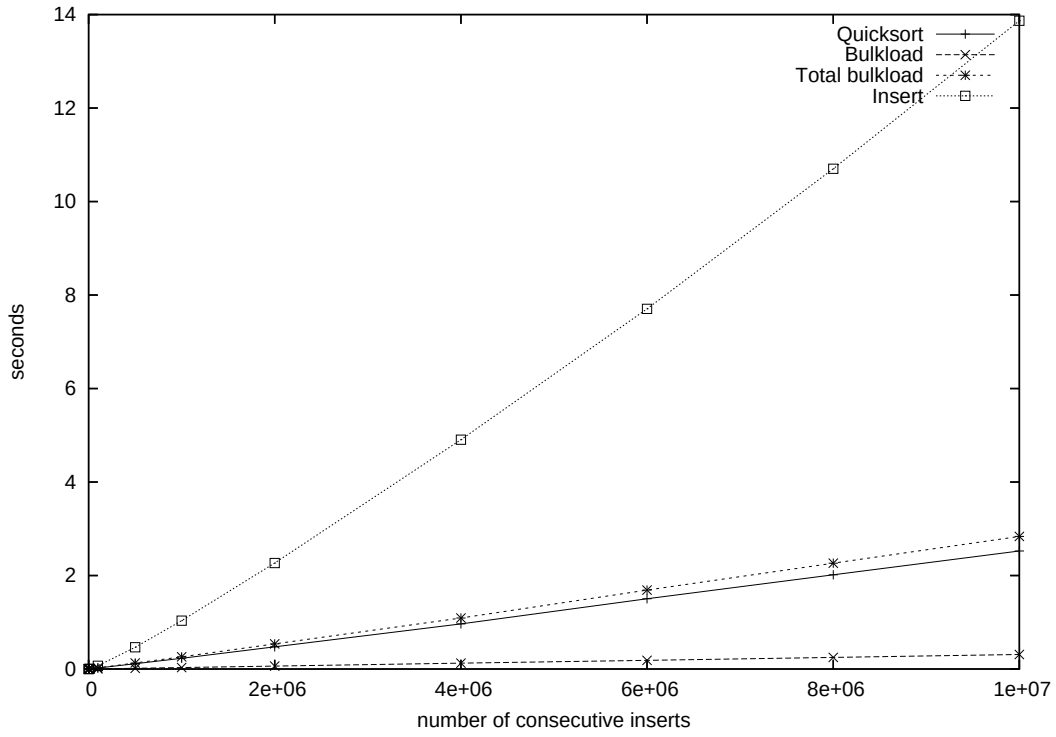
Sequence of  10000000  inserts:
Total time: 13.868
```

Disse resultatene viser helt tydelig at bulkload lønner seg, selv om dataene må sorteres først. Tiden det tar å utføre en sekvens av insert øker betydelig fortere med antall elementer som skal settes inn enn tiden for å sortere dataene og sette dem inn i treet med bulkload.

En annen ulempe med bulkload er at etter innsettingen er alle bladnodene fylt opp 100% (med mulig unntak av den siste bladnoden). Dette fører til at nesten alle innsettinger etter en bulkload vil føre til splitting av noder. Tiden for en enkelt insert er uansett så liten at dette har lite å si, spesielt med tanke på hvor mye som blir spart ved bruk av bulkload i forhold til en sekvens av insert. Bruken av bulkload er spesielt nyttig i Rindex, som ved oppstart skal laste inn alle data fra disk, for så å kunne ta i mot frittstående oppdateringer senere.

5.3.2 Find

Søk i treet foregår på akkurat samme måte som i et vanlig B^+-tre. Når nøkkelen er funnet, returneres en iterator som tillater iterering både forover og bakover. Dette gjør det mulig å finne f.eks alle like elementer eller alle elementer som er større enn det elementet det blir gjort et søk på. Hvis det fins flere like nøkler, er det den første nøkkelen som er funnet som blir returnert (med andre ord den som er lagret lengst til "venstre" i treet). Hvis nøkkelen ikke finnes, returneres det nærmeste elementet som er større. Hvis det heller ikke finnes noen større elementer, returneres det største elementet som



Figur 5.1: Kjøretider for innsetting av elementer med de to innsettingsvariantene. Total bulkload er tiden for bulkload og quicksort lagt sammen.

er lagret. Iteratoren har operasjoner for å flytte den interne pekeren framover eller bakover, i tillegg til å kunne returnere nøkkelen og data-elementet iteratoren peker på.

5.3.3 Insert

Ved innsetting gjøres først et søk etter det elementet som skal settes inn. Når noden det skal settes inn i er funnet, sjekkes det om det er plass til et nytt element i denne. Tre forskjellige tilfeller er mulige her. Det første, og enkleste, er hvis det er plass i noden. Det nye elementet settes inn, og vi er ferdige. Hvis det ikke er plass i noden, men det er plass til flere noder i nodegruppen, er det nok å splitte noden. Dette gjøres ved å "flytte" (altså kopiere) nodene med større elementer i samme nodegruppe bortover for å lage plass til en ny node ved siden av den fulle. Halvparten av elementene kopieres så over til den nye noden, og det nye elementet kan nå settes inn. Det siste, og vanskeligste, tilfellet er hvis det ikke bare er fullt i noden det skal settes inn i, men i nodegruppen den tilhører også. Da må nodegruppen splittes, noe som egentlig betyr at foreldrenoden må splittes. Ved splitting av en nodegruppe allokeres en ny nodegruppe, og halvparten av nodene i den gamle nodegruppen kopieres over i den nye. For å gjøre nettopp disse tilfellene enklere er insert implementert rekursivt. En peker til den nye nodegruppen bli returnert av den rekursive metoden, og denne kan da settes inn som barn etter å ha splittet den interne foreldrenoden. Denne rekursive metoden gjør det enkelt å utføre splitter oppover i treet hvis dette er nødvendig. Hvis rota må splittes, allokeres det en nodegruppe som vil inneholde to noder. Innholdet av den gamle rota blir fordelt på disse to nodene, og den nye nøkkelen blir satt inn på riktig sted nå som det er plass. En ny rot blir så opprettet med peker til denne nye nodegruppen.

5.3.4 Delete

Delete-operasjonen tar en iterator som parameter og sletter elementet iteratoren peker på. Fordi sammenslåing av noder er tungvint, er det vanlig å implementere delete som "lazy delete" i slike trær. Dette kan forsvares med at en database har en tendens til å vokse, ikke krympe. "Lazy delete" kan føre til brudd på regelen om at alle noder i et B^+-tre skal være minst halvfulle, siden vi bare sletter elementer i bladnodene uten å slå sammen noder. Allikevel er det rimelig å anta at det snart vil bli utført insert operasjoner som igjen fyller opp disse nodene til fyllingskravet. Jeg har av disse grunnene valgt å implementere delete som "lazy delete".

5.4 Øvrige operasjoner

I noen tilfeller er det et behov å kunne iterere over elementene i treet fra begynnelsen eller slutten, f.eks hvis man trenger å iterere over alle elementer i treet. Av denne grunn har jeg implementert to funksjoner for dette, first og last. De returnerer en iterator til henholdsvis første og siste element i treet.

Kapittel 6

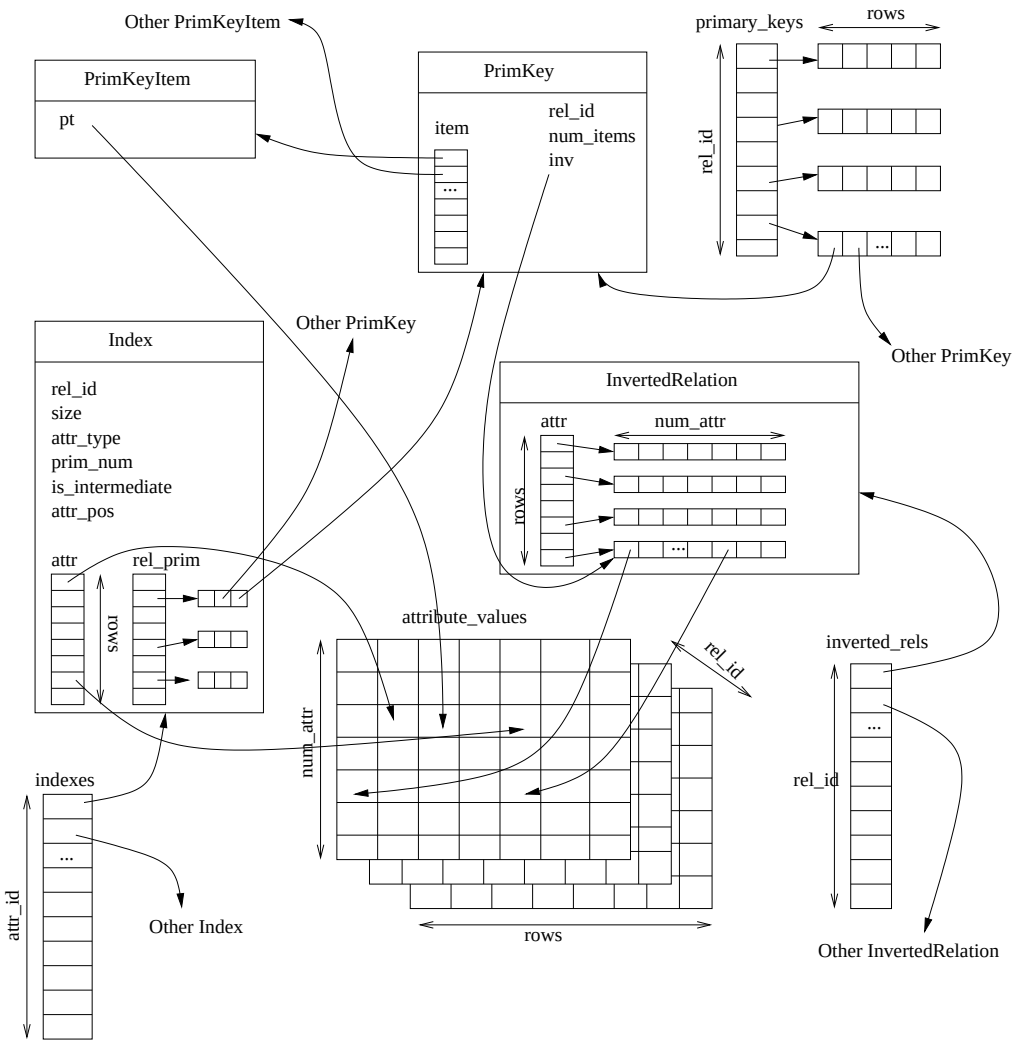
Redesign av indeksstrukturen

For å kunne gjøre bruk av CSB$^+$-trær til å lagre de indekserte verdiene må indeksstrukturen[1] endres noe. Dette er i hovedsak fordi de sekvensielle indeksene er lagret direkte i indeksstrukturen, og ikke som en egen datastruktur med operasjoner. I tillegg må alle indeksoperasjonene som f.eks seleksjon og equi-join skrives om for bruk med den nye indeksstrukturen og med hensyn på den nye datastrukturen for de indekserte verdiene. Den nye indeksstrukturen må også ta hensyn til at det nå skal kunne gjøres oppdateringer. Det er ikke bare selve indeksene, i form av oppdateringer på CSB$^+$-trærne, som må kunne oppdateres, men også de inverterte indeksene. Første avsnitt av dette kapittelet beskriver den nye strukturen for indekser og inverterte indekser, mens andre avsnitt tar for seg reimplementasjonen av indeksoperasjonene. I det tredje avsnittet beskrives de nye indeksoperasjonene, som er relatert til oppdateringer.

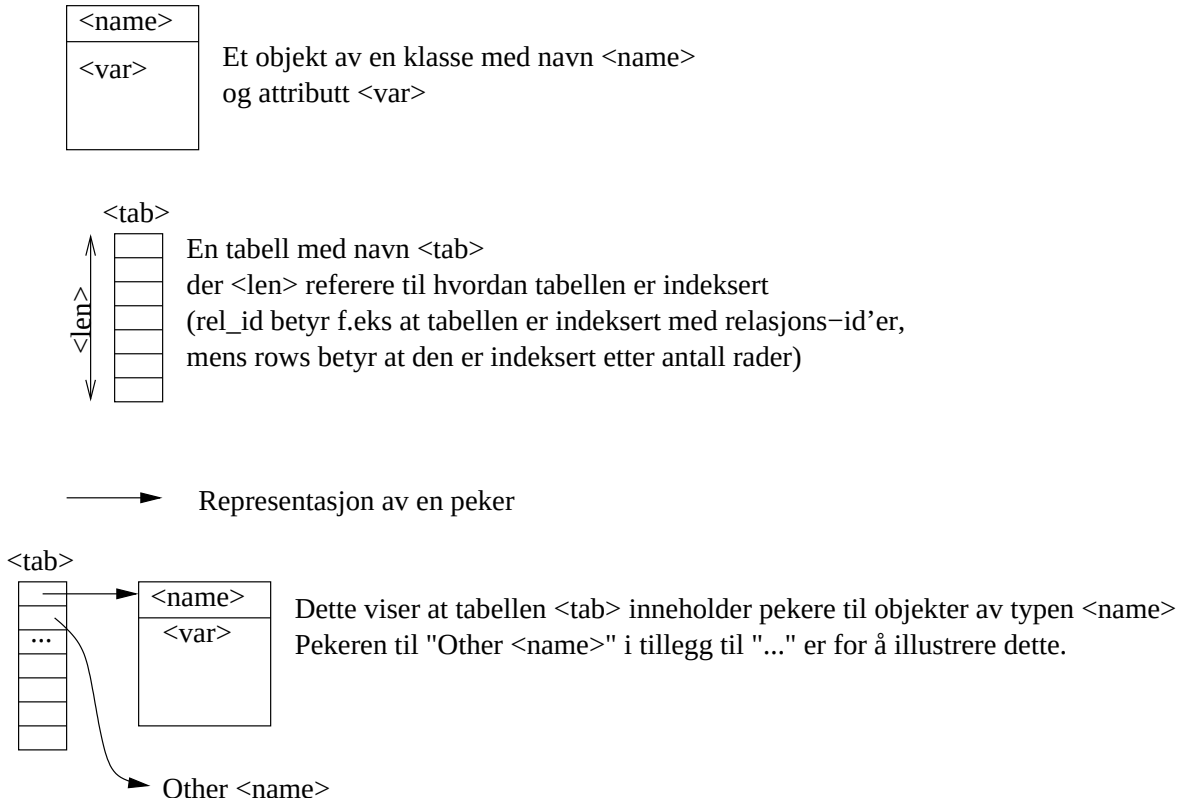
6.1 Indeksstruktur

For å bedre kunne forklare endringene i forhold til den gamle indeksstrukturen er det vist en skisse over denne i figur 6.1 på neste side. Figur 6.2 på side 25 viser en tegnforklaring til denne skissen. Alt som har med den sekvensielle lagringen av de indekserte verdiene å gjøre må selvsagt fjernes fra dette designet. Dette innebærer at tabellene "attr" og "rel_prim" i Indexklassen, som lagrer henholdsvis pekere til attributtverdier og primærnøkler, fjernes. I tillegg fjernes den tredimensjonale tabellen som sørger for at alle attributtverdier er lagret sekvensielt i minnet, da denne er overflødig i det nye designet. Disse delene byttes ut med en peker til et CSB$^+$-tre i Indexklassen. Attributtverdiene er nå lagret i noder i slike trær. Bladnodene i et CSB$^+$-tre lagrer nøkkel/data par, og i denne sammenhengen vil da nøkkelen

[1] Med indeksstruktur mener jeg strukturen rundt lagringen av alle indeksene og de inverterte indeksene.



Figur 6.1: Indeksstruktur i Rindex 0.1.1



Figur 6.2: Tegnforklaring til fig 6.1 og fig 6.3

være attributtverdien, og dataelementet vil være en peker til en tabell med pekere til primærnøkkel-elementer (PrimKey objekter)[2].

Med disse endringene er indeksstrukturen lagt til rette for bruk av CSB$^+$-trær, men det er fremdeles noen problemer som gjenstår. Den inverterte indeksstrukturen er den samme som før, og som forklart i kapittel 4, byr denne på noen problemer ved oppdateringer, og spesielt ved sletting av tupler. Det viser seg imidlertid at den inverterte indeksstrukturen kan forenkles betydelig. For det første er primary_keys tabellen fullstendig overflødig. Alle primærnøkkelobjektene kan nåes via indeksene, og tabellen brukes kun ved initialisering og sletting. I tillegg er InvertedRelation klassen kun et unødvendig abstraksjonsnivå. Denne brukes kun via inv-pekeren i PrimKey objektene, og denne brukes som en tabell. Med andre ord, det er unødvendig å først lage tabeller for alle tuplene i et InvertedRelation objekt, for så å sette pekere til disse tabellene i PrimKey objektene, når det allikevel kun er ett PrimKey objekt for hver slik tabell. Ved å fjerne InvertedRelation klassen forsvinner også behovet for inverted_rels tabellen. Den nye indeksstrukturen er skissert i figur 6.3 på neste side. De inverterte indeksene aksesseres på akkurat samme måte som før, men det er nå lettere å slette tupler fordi de ikke lenger er sekvensielt lagret i en annen tabell.

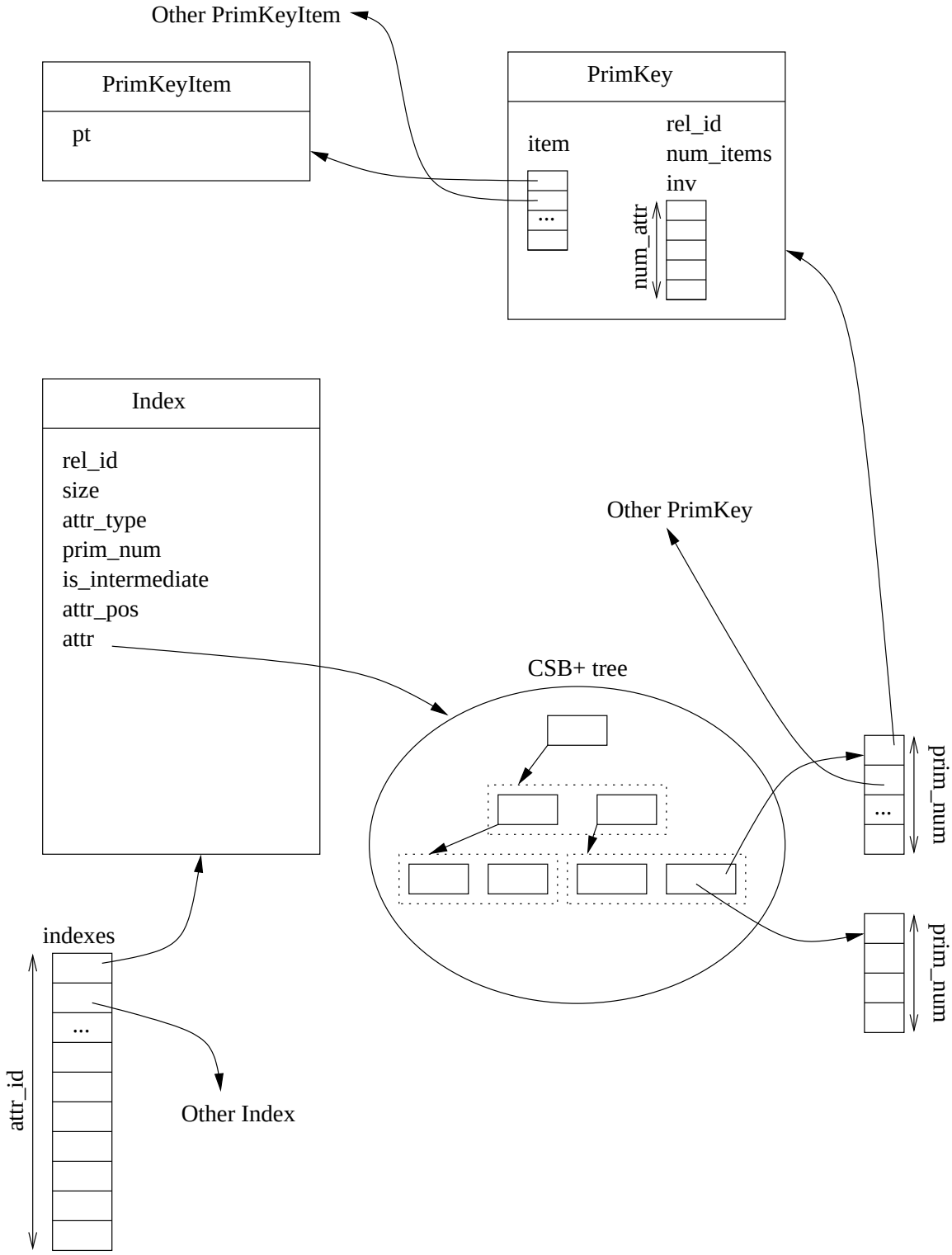
6.2 Indeksoperasjoner

Med ny indeksstruktur må alle indeksoperasjonene reimplementeres. Dette avsnittet beskriver de nye implementasjonene av indeksoperasjonene som var tilgjengelige i Rindex 0.1.1. For informasjon om hva de forskjellige operatorene brukes til, se 2.1.2. Felles for alle indeksoperasjonene er at de, som før, lager en ny midlertidig indeks. De midlertidige indeksene har identisk struktur med de vanlige indeksene.

6.2.1 Seleksjon

Det er seks forskjellige seleksjonsfunksjoner, en for hver seleksjonsoperator (lik, ulik, mindre enn, mindre enn eller lik, større enn og større enn eller lik). Å utføre en seleksjon på en indeks innebærer å hente ut elementene i det tilhørende CSB$^+$-treet som tilfredsstiller seleksjonskravet, og sette disse inn i et nytt CSB$^+$-tre. Dette nye treet vil da tilhøre den midlertidige indeksen som blir returnert. Seleksjonen utføres ved å gjøre et søk i treet etter verdien det skal sammenlignes med. Deretter brukes iteratoren til å hente ut alle

[2]Grunnen til at en indeks kan ha flere primærnøkler, er at etter en join mellom to eller flere relasjoner må indeksen som inneholder attributtverdiene det er gjort en join på, kunne peke tilbake til de inverterte indeksene til alle relasjoner som er involvert i joinen. Siden dette gjøres via primærnøkler, må indeksene kunne ha pekere til flere PrimKey objekter.



Figur 6.3: Ny indeksstruktur

elementer som passer til seleksjonskriteriet. Elementene blir så satt inn i det nye treet ved hjelp av bulkload-funksjonen, beskrevet i kapittel 5.3.1.

6.2.2 Equi-join

En equi-join er en join med et spesifisert join-attributt. Equi-join algoritmen i denne implementasjonen ligner på den gamle, men er tilpasset bruken av CSB$^+$-trær. Algoritmen består av to løkker, der den ytterste løkken itererer over den minste indeksen (den med færrest verdier/tupler), og den innerste itererer over den største indeksen. To viktige egenskaper ved CSB$^+$-trær brukes for å gjøre algoritmen effektiv.

Den første av disse egenskapene er at verdiene er sortert. Dette innebærer at den innerste løkken ikke behøver å iterere over alle verdiene i indeksen. Den kan stoppes når vi finner en verdi som er større enn den vi sammenligner med. Løkken kan så startes igjen fra dette stedet for neste verdi i den minste indeksen (den ytterste løkken), eller fra det forrige startstedet for løkken hvis den neste verdien i den andre indeksen er lik den før seg. Så langt er det ingen reelle forskjeller mellom denne algoritmen og den som er brukt i Rindex 0.1.1.

Den andre egenskapen ved CSB$^+$-trær som kan gjøre nytte her, er muligheten til å gjøre et raskt søk på en verdi. Denne egenskapen kan brukes til å unngå unødvendig iterering over den største indeksen. Ved hver nye runde i den ytterste løkken sjekkes det om den gjeldende posisjonen for starten på itereringen i den innerste løkken har en verdi som er mindre, lik eller større enn den verdien som skal sammenlignes med. Hvis verdien er mindre, så gjøres et søk i treet med find-funksjonen i steden for å gjøre et lineært søk via iteratoren. På denne måten kan man raskt finne ut om det finnes noen like verdier, og få tak i posisjonen til disse.

6.2.3 Union

Unionsoperatoren eksekveres på grunnlag av en `OR` i spørringen. Resultatet av denne operatoren skal derfor bestå av alle tuplene fra begge indeksene som er involvert, men ingen av tuplene skal forekomme mer enn en gang. Algoritmen utnytter CSB$^+$-trærnes egenskap om at verdiene er sorterte, og er dermed veldig lik algoritmen som er benyttet i Rindex 0.1.1.

Algoritmen er en standard merge som fletter de to indeksene og kaster duplikater. Fordi dataene kun er sortert på nøkkel må duplikater finnes ved lineært søk gjennom tupler med like nøkler i indeksen.

6.2.4 Projeksjon

Projeksjon utføres akkurat som før, med den forskjellen at det nå benyttes en iterator til å løpe igjennom alle verdiene som skal skrives ut. For å optimalisere selve utskriften av data er projeksjonen også endret til å skrive ut ett

og ett tuppel i steden for én og én verdi. Det vil si at verdiene for et tuppel samles opp i et buffer før de skrives ut. I tillegg er bruken av C++' utskriftsrutine *cout* byttet ut med C's *printf*, som er mer effektiv. Disse endringene ble gjort etter å ha sett hvor mye I/O-håndteringen til projeksjonen påvirket resultatene når store datasett ble skrevet ut. Dette er nærmere beskrevet i kapittel 8.2.

6.2.5 Substitusjon

Substitusjonsalgoritmens oppgave er å bytte ut en indeks med en annen indeks der tuplene er de samme, men attributtet som er indeksert er byttet ut. Algoritmen itererer over alle verdiene i indeksen som skal byttes ut, og bruker de inverterte indeksene til å hente ut verdiene til resultatindeksen. Fordi den inverterte indeksstruktur er den samme som den som er brukt i Rindex 0.1.1, dog med noen forenklinger, vil algoritmen være nærmest identisk med substitusjonsalgoritme 4 beskrevet i [3]. Kompleksiteten vil derfor også være den samme. Etter å ha hentet ut alle de nye verdiene må disse sorteres. Dette gjøres med quicksort før de sorterte verdiene settes inn i CSB^+-treet til resultatindeksen ved hjelp av bulkload. Denne innsettingen i et tre på slutten av algoritmen er det eneste som øker kompleksiteten i forhold til den gamle substitusjonsalgoritmen.

6.3 Oppdateringer

Som beskrevet i kapittel 4, innebærer oppdateringer på indeksene oppdateringer på metadataene i tillegg til selve indeksstrukturen. Fordi metadatastrukturen benyttet i denne implemententasjonen er identisk med den brukt i implementasjonen beskrevet i kapittel 4, vil denne ikke bli nevnt ytterligere her. Dette avsnittet beskriver kun operasjonene som blir utført på indeksene ved oppdateringer, diskoppdateringer er beskrevet i kapittel 7.

6.3.1 Create

Når en ny relasjon blir opprettet, blir det initialisert objekter av typen Index for alle de nye attributtene. Dette innebærer at tabellen "indexes" vokser og fylles opp av pekere til Index-objekter. Alle disse nye Index-objektene (se figur 6.3) vil ha pekere til et tomt CSB^+-tre. Disse trærne vil bli fylt opp ved senere innsettinger i den nye relasjonen.

6.3.2 Drop

Når en relasjon skal droppes, må alle objekter som hører til denne relasjonen slettes. Figur 6.3 viser hvor mange forskjellige typer objekter som er forbundet med hver indeks, altså hvert attributt i en relasjon. Det er ett Index ob-

jekt for hvert attributt og ett PrimKey objekt for hvert tuppel i relasjonen[3]. For å få slettet alle disse objektene må man først iterere over alle verdier i ett av Index-objektene for å få slettet alle PrimKey- og PrimKeyItem-objekter. Deretter kan alt minnet allokert til CSB$^+$-trærne frigis, og til slutt slettes selve Index-objektene. Fordi attributt-id'er ikke gjenbrukes, kan ikke "indexes" tabellen reduseres i størrelse. I steden fylles de ubrukte feltene i tabellen ut med NULL-verdier.

6.3.3 Insert

Alle tupler har et tilhørende PrimKey objekt, så innsetting av et nytt tuppel betyr opprettelse av et nytt PrimKey objekt, med tilhørende PrimKeyItem objekter for hvert attributt som er med i primærnøkkelen. Verdiene i det nye tuppelet fylles inn i "inv" tabellen i PrimKey objektet. Deretter må alle verdiene i det nye tuppelet settes inn i sine respektive indekser. Dette gjøres ved insert-operasjoner på CSB$^+$-trærne med pekeren til det nye PrimKey objektet som data-element[4].

6.3.4 Update

En `UPDATE`-setning innebærer oppdatering av verdiene til visse attributter på visse tupler i en relasjon. Hvilke tupler som skal oppdateres, bestemmes av en eller flere seleksjoner som eksekveres før update-funksjonen på indeksene kalles. Denne funksjonen får da inn en indeks som inneholder de tuplene som skal oppdateres, i form av pekere til PrimKey objekter lagret i CSB$^+$-treet til Index-objektet, slik som vist i figur 6.3. Verdien til ett attributt i et gitt tuppel må oppdateres to steder i indeksstrukturen. Det ene stedet er nøkkelen som brukes i CSB$^+$-treet, og det andre stedet er i "inv" tabellen i PrimKey objektet til tuppelet. PrimKey objektet brukes til å finne elementet som skal oppdateres i Index-objektene til hvert attributt. Et søk på den gamle verdien i CSB$^+$-treet, etterfulgt av et lineært søk etter det riktige tuppelet (PrimKey objekt), gir elementet som skal slettes fra treet. Dette slettes så ved kall på delete-rutinen til treet. Etter at alle de gamle verdiene er slettet, går algoritmen igjennom input-indeksen på nytt, henter ut PrimKey objektene og setter disse inn igjen i indeksene som skal oppdateres, sammen med de nye verdiene. Parallelt med dette oppdateres alle attributtene i "inv" tabellen.

6.3.5 Delete

Delete ligner på update i den forstand at en eller flere seleksjoner har gitt oss de tuplene som skal slettes. Algoritmen ligner derfor på den for update,

[3] Disse er de samme for alle attributter i ett og samme tuppel.

[4] Egentlig er det en peker til en tabell av størrelse én, der det eneste feltet inneholder pekeren til PrimKey objektet som blir satt inn. Dette er på grunn av kompabilitet med de midlertidige indeksene, som kan ha pekere til flere PrimKey objekter.

med den forskjellen at ingen verdier blir satt inn igjen i indeksene etter slettingen, i tillegg til at attributtverdiene i "inv" tabellen slettes, i steden for å oppdateres. Dette er en betydelig enklere prosedyre enn den som er beskrevet i kapittel 4 angående delete i implementasjonen med sekvensielle indekser. Dette er fordi de delene av den inverterte indeksstrukturen som skapte vanskeligheter for sletting av tupler i den implementasjonen, er fjernet i den nye indeksstrukturen.

Kapittel 7

Oppdateringer på disk

Så langt har vi kun diskutert endringer av indeksene i minnet. For at disse endringene ikke skal kunne bli borte, må de også skrives til disk. Siden Rindex er bygd for å fungere oppå en allerede eksisterende diskdatabase, er det enkelt og naturlig å la denne databasen ta seg av logging av endringer. Ved å sende endringene til diskdatabasen før endringene gjøres på indeksene i Rindex, sikrer vi at Rindex alltid gjenspeiler den underliggende databasen 100%. Hvis det ikke er mulig å utføre endringene på diskdatabasen, vil heller ikke Rindex' indekser bli oppdatert. Dette gjør at hvis den underliggende databasen er nede, kan Rindex fortsette å kjøre og ta imot spørringer, men endringer vil bli forkastet. Det er verdt å nevne at hvis endringer i databasen ble gjort direkte i den underliggende databasen, uten å gå via Rindex, ville Rindex' indekser måtte regenereres for å igjen kunne gjenspeile databasen.

Rindex støtter i utgangspunktet to forskjellige databaser, CSV[1]-filer og Oracle. Fordi CSV-støtten i stor grad er hardkodet på grunnlag av et par testfiler, og kun nyttig for initiell testing av spørringer i Rindex, har jeg valgt å ikke utvide denne med støtte for oppdateringer. Resten av dette kapittelet beskriver hvordan støtte for oppdateringer er lagt til i Oracle-frontenden.

7.1 Interface

Endringer til Oracle sendes ved hjelp av SQL-oppdateringer. Fordi forskjellige DBMS'er har implementert SQL forskjellig, og fordi Rindex skal kunne støtte flere typer DBMS'er, er det viktig at alt som har med SQL-generering å gjøre skjer i OracleFrontend klassen beskrevet i kapittel 5.3 i [3]. Da kan grensesnittet mot de andre delene av Rindex, definert i DBFrontend, være generelt slik at støtte for andre DBMS'er enkelt kan legges til.

Grensesnittet for create trenger navnet på relasjonen, en liste over attributter med tilhørende typer og en liste med navnene til primærnøkkelattributtene. Drop klarer seg med navnet på relasjonen, mens insert må ha

[1] Comma Separated Values

en liste over attributtnavnene med en tilhørende liste av verdier i tillegg til relasjonsnavnet.

Når det gjelder update og delete, er det ikke fullt så enkelt å lage et enkelt grensesnitt. Disse spørringene kan nemlig ha et `WHERE`-ledd som forteller hvilke rader som skal oppdateres eller slettes. Hvordan kan betingelsene til `WHERE`-leddet enkelt sendes som parametre til en funksjon? Det er vanskelig å finne en enkel måte å gjøre dette på, og et mer fundamentalt spørsmål er hvordan betingelsene kan gjøres om fra slik de er lagret internt i Rindex, til SQL. Betingelsene til `WHERE`-leddet er i Rindex lagret i parsetreet, som gjøres om til en spørreplan før eksekveringen. Siden skrivingen til disk foregår som en del av eksekveringen, er allerede konverteringen av treet til en spørreplan utført. Fordi det er vanskeligere å finne tilbake til SQL-form fra spørreplanen enn fra det initielle parsetreet, kan det være lønnsomt å bruke parsetreet i steden. Siden parsetreet blir slettet ved konvertering til spørreplanen, må konverteringen av parsetreet til et SQL `WHERE`-ledd skje før konverteringen til en spørreplan. Dette er den løsningen som er valgt. Før kall på funksjonene som tar seg av optimalisering og generering av spørreplan, kalles en funksjon i DBFrontend som traverserer treet og lager en streng med `WHERE`-leddet. Denne lagres så i OracleFrontend klassen, til senere bruk. Dette fjerner behovet for å sende betingelsene som parametre i det hele tatt.

En annen løsning kunne være å ikke finne tilbake til det opprinnelige `WHERE`-leddet, men benytte det faktum at Rindex lagrer indekser, og dermed vet hvilke rader som skal oppdateres. Da kunne primærnøkkel-verdiene til alle rader som skal oppdateres brukes til å generere en serie med `UPDATE`-spørringer. Dette er en interessant løsning det kan være verdt å teste ut effektiviteten av, men jeg har valgt å ikke bruke denne løsningen, da det kan generere et meget stort antall `UPDATE`-spørringer, og effektiviteten av dette er tvilsom. Grunnet tidspress har jeg ikke testet ut dette.

7.2 Datatyper

Rindex støtter fem forskjellige datatyper: tekst, 32-bits heltall, 64-bits heltall, 32-bits flyttall og 64-bits flyttall. De eneste datatypene til Oracle som er støttet av Rindex 0.1.1 er `VARCHAR2`, `CHAR` og `NUMBER`. `VARCHAR2` og `CHAR` kan lagre tekster på opptil 255 tegn, mens `NUMBER` er en fellesbetegnelse i Oracle for alle typer tall, det vil si både heltall og flyttall. `NUMBER` kan ta to parametre, precision og scale (se [7]). Forskjellige kombinasjoner av disse forteller Oracle hvilke typer tall som er lovlig for et attributt, og hvordan disse skal vises for brukeren. Precision oppgis som et tall mellom 1 og 38, mens scale må være mellom 1 og 127. Precision angir maks antall siffer som er lovlige for attributtet, mens scale angir hvor mange desimaler som skal vises ved output. Scale har altså ingen påvirkning på hvordan tallet lagres, og begrenser heller ikke inputen. Scale er kun et mål på hvor tallet skal rundes

av ved utskrift.

Rindex 0.1.1 konverter `VARCHAR2` og `CHAR` til tekst, og `NUMBER` til integer (32-bits heltall), uten å ta hensyn til precision og scale. Jeg antar at dette er gjort under antagelsen av at alle testdata skulle være heltall. Jeg har valgt å endre på dette, og også ta hensyn til precision og scale ved konvertering til Rindex' datatyper. Det vanskelige med dette er at Oracle ikke begrenser tallene på samme måte som C++. Oracle begrenser tallene med antall siffer, mens C++ datatyper begrenser tallene med hvor store tall som kan lagres med et gitt antall bits. Det enkleste ville være å konvertere alle `NUMBER` typer til double, som vil kunne lagre alle tall som kan lagres i `NUMBER` typen til Oracle. Dette er dog ikke hensiktsmessig. Grunnen til dette er at nodestørrelsen i CSB^+-trærne er konstant uansett nøkkelstørrelsen. Hvis nøklene er double-verdier, vil disse trenge 8 byte lagringsplass, og det blir plass til færre verdier i en node enn hvis nøklene er på 4 byte. Dette gjør at trær med 8 byte nøkler kan bli noe dypere enn trær med 4 byte nøkler. Det er derfor mest hensiktsmessig å ha nøkler på 4 byte der dette er mulig. Jeg har derfor valgt å gjøre noen approksimasjoner. Hvis precision er satt, men ikke scale, så er kun heltall tillatt. Dermed kan int og long brukes. Int kan brukes hvis precision er 10 eller mindre, mens long kan brukes hvis precision er 19 eller mindre. Tall med flere enn 19 siffer kan ikke representeres av long. Opp til 38 siffer kan lagres i en float, derfor har jeg valgt å bruke float for de største heltallene. Når det gjelder flyttallene (der scale er satt), så bruker jeg float hvis scale er mindre enn eller lik 38.

Hvis scale er større enn dette, så betyr det at scale er større enn precision. Dette betyr i Oracle at tallet må være mellom 0 og 1, og precision vil da angi hvor mange signifikante desimaler som kan lagres. Med andre ord, $scale - precision$ gir antall nuller etter komma, før de signifikante desimalene. Disse tallene kan ikke lagres i en float, men må lagres med double for å ikke miste presisjon. `NUMBER` kan også angis uten precision eller scale. I dette tilfellet er `NUMBER` ekvivalent med en float.

Når det gjelder konvertering den andre veien, hvis en relasjon opprettes via Rindex, så lagres tekst som `VARCHAR2(255)`, og tall konverteres på tilsvarende måte som over. Int lagres som `NUMBER`(10), long som `NUMBER`(19) og float som `NUMBER`. Double kan representere større tall enn `NUMBER` kan, men konverteres likevel til `NUMBER`. Dette vil medføre tap av presisjon, men er den enkleste løsningen. I praksis betyr det at ved bruk av Oracle som underliggende database støtter ikke Rindex 64-bits flyttall.

Kapittel 8

Ytelsestester av spørringer

8.1 Formål og testmetode

Rindex 0.1.1, som ikke støttet oppdateringer, har gjennom to forskjellige implementasjoner av støtte for oppdateringer utviklet seg til to nye versjoner, heretter kalt Rindex 0.2 og Rindex 0.3. Rindex 0.2 bruker den samme indeksstrukturen som Rindex 0.1.1, med kun noen små endringer for å muliggjøre oppdateringer, mens Rindex 0.3 har fått helt ny indeksstruktur. For å finne ut om endringene som er innført i disse to nye versjonene har påvirket ytelsen til Rindex, er det nødvendig å gjøre empiriske tester.

8.1.1 Testdatabaser

Testdatabasene som er brukt er nummerdatabasen og filmdatabasen. Nummerdatabasen er en database bestående av tre relasjoner, X, Y og Z som inneholder heltall. De tre tabellene har tre attributter hver, og både Y og Z har fremmednøkler til X. Databasen er beskrevet nærmere i kapittel 11.3 i [12]. Nummerdatabasen slik den ble brukt til ytelsestestene av Rindex 0.1.1, var noe liten. Relasjonene X, Y og Z hadde henholdsvis 1000, 10 000 og 30 000 tupler. Disse størrelsene er blitt økt med en faktor på 10, til 10 000, 100 000 og 300 000, for å få et datasett som vil differensiere bedre mellom de forskjellige implementasjonene.

Filmdatabasen er et utsnitt av databasen IMDB [4], og inneholder data om 15 595 filmer og personene som er involvert i disse. Den største relasjonen (Participation) har 232 598 tupler. Databasen er beskrevet nærmere i [9].

8.1.2 Hva skal testes?

For å finne ut hvilken av de to nye versjonene av Rindex som har best ytelse, må disse testes mot hverandre. I tillegg må de testes mot Rindex 0.1.1 for å finne ut om ytelsen er endret i forhold til denne. Det er selvsagt ønskelig

at ytelsen til Rindex-plattformen ikke er blitt dårligere med innføringen av støtte for oppdateringer.

Fordi nummerdatabasen brukt her er 10 ganger større enn nummerdatabasen brukt i testene i [12], kan det også være interessant å kjøre noen tester for å se hvordan et større datasett påvirker forskjellene i ytelse mellom Oracle brukt med Rindex som overbygning og fullt indeksert Oracle.

Det er ønskelig å teste alle aspektene ved SQL som er støttet i Rindex, seleksjoner, join, projeksjon og kombinasjoner av disse. Fordi alle disse aspektene er godt dekket av spørringene brukt i ytelsestestene beskrevet i [12], vil de samme spørringene bli brukt her. Spørringene er beskrevet nærmere i avsnitt 8.2.

8.1.3 Testmetode

Gjennomføring av testene

For å effektivisere testingen brukes et skript. Skriptet starter Rindex-versjonen som skal testes, og kjører alle testspørringene et gitt antall ganger. For å få godt nok statistisk tallmateriale er dette antallet satt til 100.

Klientskriptet som ble utviklet sammen med Rindex 0.1.1 brukes for å starte og stoppe Rindex, og til å sende spørringene til Rindex. Fordi det nå er flere versjoner av Rindex som skal testes mot hverandre, er det lagt til en opsjon i klientskriptet som gjør at det kan starte den angitte versjonen av Rindex. For at dette skal virke må det lages en kopi av skriptet som setter opp omgivelsene for Rindex og starter serveren for hver Rindex-versjon. Disse må hete rindex-<versjonsnr>, og ligge tilgjengelige for kjøring i den katalogen testene skal kjøres fra. Hvert slikt skript starter da den riktige Rindex-versjonen fra den katalogen den ligger i.

Spørringer til Oracle sendes ved hjelp av sqlplus. Bruken av klientskript for å kommunisere med Rindex og Oracle gjør at eksekveringstidene som måles vil inkludere tiden brukt på kommunikasjon mellom klientskriptet og serveren. Denne tiden antas å være neglisjerbar i forhold til eksekveringstiden til serveren, og vil derfor bli ignorert.

GNUs versjon av time brukes til å måle eksekveringstiden. For å slippe skriving til skjerm startes Rindex med opsjonen -q, som sørger for at all output til standard ut skrives til /dev/null. Tilsvarende sendes outputen fra sqlplus til /dev/null.

Kommandoen for å starte Rindex er:

```
client.plx -v <versjonsnr> -s -- -q -m oracle -u <brukernavn> \
-p <passord>
```

For å sende testene til Rindex og Oracle brukes følgende kommandoer:

```
/usr/bin/time -f "%e" -o <datafil> -a -- client.plx -s <filnavn>
/usr/bin/time -f "%e" -o <datafil> -a -- sqlplus -S \
```

```
<brukernavn>/<passord>@ififorsk @<filnavn> < stop.tmp \
 > /dev/null
```

Parametrene til /usr/bin/time gjør at eksekveringstiden, i sekunder, blir lagt til på slutten av filen <datafil>. Filen stop.tmp er en fil som inneholder kommandoen *quit*. Denne er nødvendig for at sqlplus skal avslutte. Opsjonen -s sørger for dette i klient-skriptet til Rindex.

Testmaskinen

Testmaskinen er en maskin dedikert til forskning, og er i tidsrommet testene ble kjørt ikke bruk av noen andre. Testene kan derfor kjøres på denne uten å bli forstyrret av annen belastning. Det er likevel mulig at det innimellom kjører cronjobber på testmaskinen. Dette kan det ikke gjøres noe med grunnet mangelen på privilegier til testmaskinen. Dette kan være årsaken til noen ekstreme outliere som av og til forekommer i testresultatene.

Maskinen har to Intel Xeon 2.4 GHz prosessorer og 2GB RAM.

Presentering av resultatene

Resultatene vil bli presentert ved hjelp av 5-nummeroppsummering og gjennomsnittsverdi. 5-nummeroppsummeringen består av minste verdi, største verdi, median, 1. kvartil og 3. kvartil. 1. kvartil er medianen til dataene som er mindre enn medianen, mens 3. kvartil er medianen til dataene som er større enn medianen. I tillegg vil dataene bli plottet ved hjelp av boksplott, som er en representasjon av 5-nummeroppsummeringen. Ekstreme outliere vil bli fjernet fra dataene, da disse mest sannsynlig skyldes ytre påvirkninger, og ikke representer reell eksekveringstid.

8.1.4 Problemer med Rindex 0.1.1

Initielle tester brukt til å teste at Rindex 0.3 ga riktig resultat på de spørringene som skulle testes, viste en feil i plattformen som er blitt overført fra Rindex 0.1.1. Denne feilen oppstår når spørringen har flere `AND`-ledd gruppert sammen med parenteser, og gjør at noen `AND`-ledd blir borte ved konverteringen av parsetreet til en eksekverbar plan. Dette medfører at ikke alle seleksjoner som skulle vært utført blir utført, og dermed kan resultatet bli galt. Feilen ble oppdaget ved testing av spørringen som ble brukt til å teste om det ville være gunstig å dytte seleksjoner i en spørreplanoptimalisering i Rindex (se kapittel 12.6 i [12]). Feilen er så alvorlig at resultatsettet returnert av Rindex var nesten dobbelt så stort som det Oracle[1] returnerer på denne spørringen. Etter å ha rettet denne feilen ble det oppnådd en vesentlig ytelsesforbedring av Rindex på denne spørringen. Patchen er implementert

[1]Både her og i resten av kapitlet skal termen "Oracle" tolkes som Oracle brukt med interne indekser og uten Rindex.

i både Rindex 0.2 og Rindex 0.3. For å oppnå rettferdig sammenligning med Rindex 0.1.1 er patchen også implementert i denne versjonen, men under nytt versjonsnummer, 0.1.2. Rindex 0.1.1 inneholder altså ikke denne patchen i testene nedenfor.

Under testing av skriptet som skulle brukes til å kjøre testene dukket det opp et annet urovekkende problem. Ytelsen til Rindex viste seg å være dårligere enn Oracle på nesten samtlige spørringer mot den utvidete nummerdatabasen. Dette gjaldt alle versjoner. Nærmere analyse viste at størsteparten av eksekveringstiden til Rindex ble brukt av projeksjonen. Dette er ikke fordi projeksjonen i seg selv er treg, men på grunn av måten Rindex håndterer I/O. Etter å ha optimalisert I/O håndteringen til Rindex, ved å blant annet gjøre færre I/O kall (skrive ut ett og ett tuppel i steden for ett og ett attributt), og bytte ut C++' cout funksjon med C's printf (som er raskere) ble ytelsen forbedret betydelig. Disse endringene er implementert i Rindex 0.2, Rindex 0.3 og Rindex 0.1.2.

Grunnet disse to problemene med tilhørende forbedringer er det fire Rindex versjoner som er testet mot hverandre. Rindex 0.1.1, Rindex 0.1.2, Rindex 0.2 og Rindex 0.3. Rindex 0.1.1 er tatt med for å vise forbedringen i ytelse grunnet de to forbedringene beskrevet her.

8.2 Testene

8.2.1 Seleksjon

Spørringene

Følgende spørringer er brukt for å teste ytelsen i utførelsen av seleksjoner:

1. ```
   SELECT *
   FROM Z
   WHERE Z1 < 49000 OR Z2 < 49000 OR Z3 < 49000
      OR Z1 > 51000 OR Z2 > 51000 OR Z3 > 51000;
   ```

2. ```
   SELECT *
   FROM Z
   WHERE (Z1 < 49000 OR Z1 > 51000)
      OR (Z2 < 49000 OR Z2 > 51000)
      OR (Z3 < 49000 OR Z3 > 51000);
   ```

3. ```
   SELECT *
   FROM Z
   WHERE (Z1 <> 1 AND Z1 <> 2)
     AND (Z2 <> 3 AND Z2 <> 5)
     AND (Z3 <> 8 AND Z3 <> 13);
   ```

4. ```
   SELECT *
   ```

```
   FROM FilmRating
   WHERE (filmid <> 1 AND filmid <> 2)
     AND (rating <> 3 AND rating <> 5)
     AND (votes <> 8 AND votes <> 13);
```

5.
```
   SELECT *
   FROM Participation
   WHERE partname <> '1' AND partname <> '2'
     AND partname <> '3' AND partname <> '5';
```

6.
```
   SELECT *
   FROM Participation
   WHERE pid <> 1 AND pid <> 2
     AND pid <> 3 AND pid <> 5;
```

De tre første av disse spørringene er mot nummerdatabasen, mens de tre siste er mot filmdatabasen. Spørring 1 og 2 gir identisk resultat, men spørring 2 er optimal i forhold til antall substitusjoner som blir utført i Rindex. Grunnen til at jeg har tatt med begge to er at det i [12] ble vist at spørring 2 hadde bedre ytelse i forhold til Oracle enn det spørring 1 hadde. Hensikten med å ta med begge er da å se om det fortsatt er en ytelsesforbedring å se ved å kjøre spørring 2 framfor spørring 1 mot Rindex, altså om substitusjonsalgoritmen i Rindex 0.3 er rask nok til å eliminere denne forskjellen eller ikke.

Spørring 1 og 2 er utformet for å teste seleksjoner med `OR`, mens de øvringe spørringene tester seleksjoner med `AND`. Spørring 4 er utformet for å være så lik som mulig spørring 3, men da altså mot filmdatabasen. Spørring 5 og 6 er med for å teste om det er noen ytelsesforskjeller i Rindex mellom strengdata-sammenligning og talldata-sammenligning. Spørringene er ment å være så like som mulig, og utføres på den største relasjonen i filmdatabasen. Spørring 5 kan ha en svakhet i denne sammenhengen, nemlig at det kun sammenlignes med ett tegn. Testen vil dermed ikke vise svakheter i Rindex angående sammenligning av lange strenger. Jeg har valgt å ikke endre denne spørringen, da hensikten med disse testene er å sammenlignde Rindex 0.3 med de andre Rindex-versjonene, og ikke med Oracle. Siden alle Rindex-versjonene håndterer strenger likt, er denne testen helt grei til dette formål.

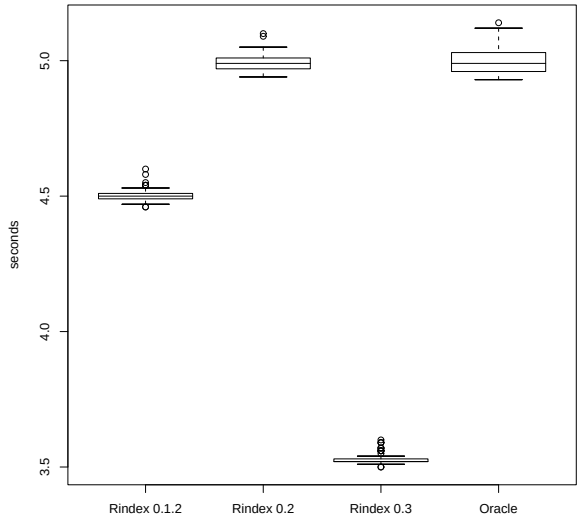
Resultatene

Spørring 1

```
SELECT * FROM Z
WHERE Z1 < 49000 OR Z2 < 49000 OR Z3 < 49000
   OR Z1 > 51000 OR Z2 > 51000 OR Z3 > 51000;
```

	Rindex 0.1.1	Rindex 0.1.2	Rindex 0.2	Rindex 0.3	Oracle
Min.	8.270	4.460	4.940	3.500	4.930
1st Qu.	8.518	4.490	4.970	3.520	4.960
Median.	8.570	4.500	4.990	3.520	4.990
3rd Qu.	8.630	4.510	5.010	3.530	5.030
Max.	8.870	4.600	5.100	3.600	5.140
Mean.	8.557	4.502	4.992	3.527	4.998

Tabell 8.1: Spørring 1



Figur 8.1: Seleksjonstest 1

Hvis vi ser på 5-nummer oppsumeringen til dataene fra den første spørringen, vist i tabell 8.1, så ser vi først at det er en klar forbedring fra Rindex 0.1.1 til Rindex 0.1.2, som skyldes I/O optimaliseringen. Rindex 0.3 viser seg å ha de beste tidene blant alle Rindex versjonene, og er også vesentlig raskere enn Oracle. Figur 8.1 på forrige side viser boksplottet til disse dataene. Rindex 0.1.1 er utelatt fra dette plottet på grunn av den store forskjellen i tider mellom denne og de andre versjonene. Det er heller ikke interessant å sammenligne Rindex 0.1.1 med noe annet enn Rindex 0.1.2, siden tidsforskjellene disse imellom kun skyldes I/O håndtering. En annen ting som er verdt å legge merke til, og som synes tydelig hvis vi ser på boksplottet, er at Rindex 0.2 og Oracle har omtrent like eksekveringstider på denne spørringen. Rindex 0.2, som altså er basert på sekvensielle indekser, har signifikant dårligere ytelse enn Rindex 0.1.2. Dette skyldes at datastrukturen vector er tregere å aksessere enn å bruke arrayer som aksesserer minnet direkte. En nærmere analyse av hva som tar tid under eksekveringen i Rindex, viser at det sekundet Rindex 0.3 sparer i forhold til Rindex 0.1.2 skyldes en raskere union-algoritme, i tillegg til at sorteringen som utføres av substitusjonsalgoritmen går raskere i Rindex 0.3.

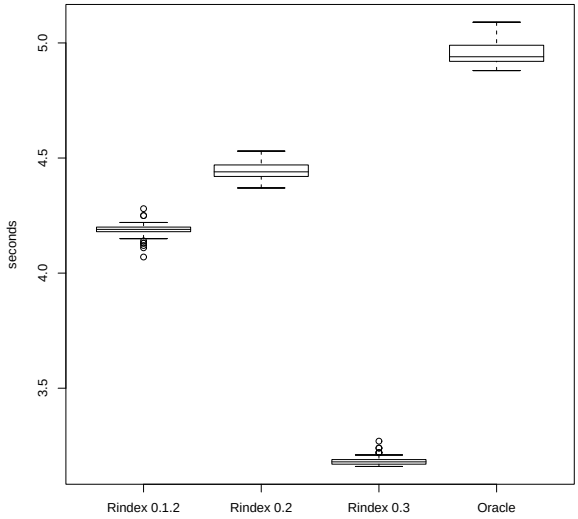
Spørring 2

```
SELECT * FROM Z
WHERE (Z1 < 49000 OR Z1 > 51000)
   OR (Z2 < 49000 OR Z2 > 51000)
   OR (Z3 < 49000 OR Z3 > 51000);
```

	Rindex 0.1.1	Rindex 0.1.2	Rindex 0.2	Rindex 0.3	Oracle
Min.	7.900	4.070	4.370	3.160	4.880
1st Qu.	8.098	4.180	4.420	3.170	4.920
Median.	8.165	4.190	4.440	3.180	4.940
3rd Qu.	8.220	4.200	4.470	3.190	4.990
Max.	8.420	4.280	4.530	3.270	5.090
Mean.	8.150	4.189	4.444	3.182	4.954

Tabell 8.2: Spørring 2

Dette er en optimalisert versjon av den første spørringen med hensyn på antall substitusjoner som blir utført av Rindex. Testen viser at det for alle versjoner av Rindex er tid å hente på å minke antall substitusjoner som blir utført. Vi ser også at Rindex 0.2 har bedre ytelse enn Oracle på denne spørringen. Rindex 0.3 er fremdeles raskest, og også denne Rindex-versjonen blir raskere med færre substitusjoner. Boksplottene for denne spørringen er vist i figur 8.2.



Figur 8.2: Seleksjonstest 2

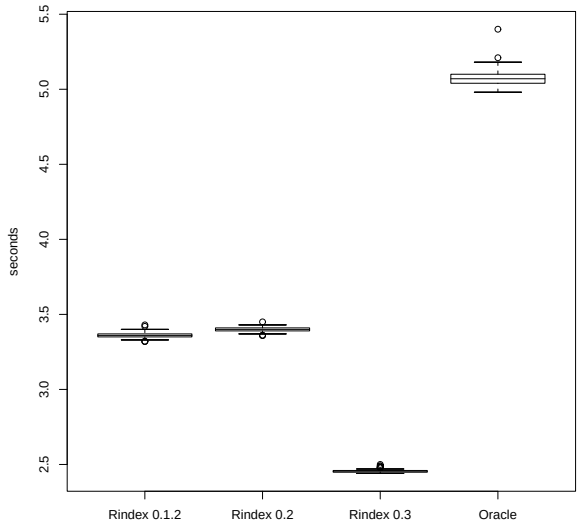
Spørring 3

```
SELECT * FROM Z
WHERE (Z1 <> 1 AND Z1 <> 2)
  AND (Z2 <> 3 AND Z2 <> 5)
  AND (Z3 <> 8 AND Z3 <> 13);
```

	Rindex 0.1.1	Rindex 0.1.2	Rindex 0.2	Rindex 0.3	Oracle
Min.	6.900	3.32	3.36	2.440	4.980
1st Qu.	7.060	3.35	3.39	2.450	5.040
Median.	7.110	3.36	3.40	2.450	5.070
3rd Qu.	7.160	3.37	3.41	2.460	5.100
Max.	7.350	3.43	3.45	2.500	5.400
Mean.	7.103	3.36	3.40	2.456	5.077

Tabell 8.3: Spørring 3

Denne spørringen viser den samme tendensen som de to første angående ytelsesforbedringen mellom Rindex 0.1.1 og Rindex 0.1.2. Også her er Rindex 0.1.1 utelatt fra boksplottet, vist i figur 8.3, grunnet den store ytelsesforskjellen. Når det gjelder forskjellen mellom Rindex 0.1.2 og Rindex 0.2, så ser vi at den er mindre her, men at Rindex 0.1.2 fremdeles er noe raskere. Grunnen til at forskjellen er mindre her er at en rekke med `AND`-operasjoner ikke



Figur 8.3: Seleksjonstest 3

medfører noe annet enn seleksjoner, mens `OR`-operasjoner krever også union av resultatsettene underveis. Dette gjør at det for Rindex 0.2 blir vesentlig færre aksesser i vektorene, noe som kan forklare hvorfor forskjellene her er mindre. Rindex 0.3 er også på denne spørringen vesentlig raskere enn de andre Rindex-implementasjonene, mens Oracle ikke kan vise like god ytelse som noen av Rindex implementasjonene, med unntak av Rindex 0.1.1. Som for de to foregående spørringene, er det forbedringen i ytelse til substitusjons-algoritmen, grunnet en raskere sortering, som er årsak til ytelsesforskjellen mellom Rindex 0.3 og Rindex 0.1.2.

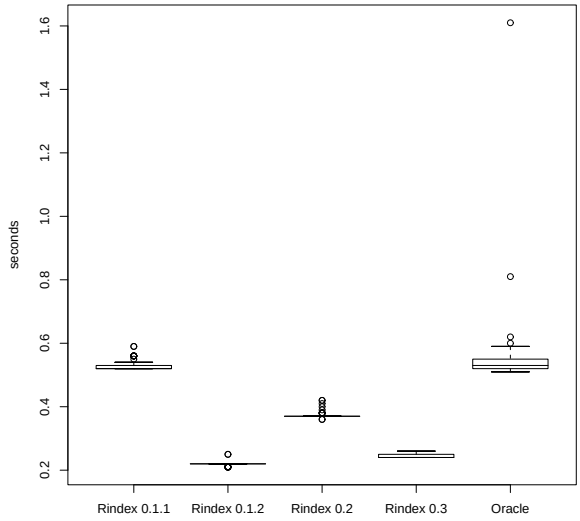
Spørring 4

```
SELECT * FROM FilmRating
WHERE (filmid <> 1 AND filmid <> 2)
  AND (rating <> 3 AND rating <> 5)
  AND (votes <> 8 AND votes <> 13);
```

Denne spørringen mot filmdatabasen var i utgangspunktet ment å skulle være sammenlignbar med den tredje spørringen mot nummerdatabasen, men når Z relasjonen er mye større enn FilmRating relasjonen, er ikke dette lenger mulig. Spørringen er nå kun med for å se på forholdet i ytelse mellom de forskjellige Rindex-versjonene. Hvis vi ser på 5-nummeroppsummeringen i tabell 8.4 og boksplottet nederst i figur 8.4, så ser vi at Rindex 0.3 ikke er

	Rindex 0.1.1	Rindex 0.1.2	Rindex 0.2	Rindex 0.3	Oracle
Min.	0.5200	0.2100	0.3600	0.2400	0.5100
1st Qu.	0.5200	0.2200	0.3700	0.2400	0.5200
Median.	0.5200	0.2200	0.3700	0.2400	0.5300
3rd Qu.	0.5300	0.2200	0.3700	0.2500	0.5500
Max.	0.5900	0.2500	0.4200	0.2600	1.6100
Mean.	0.5284	0.2194	0.3723	0.2435	0.5498

Tabell 8.4: Spørring 4



Figur 8.4: Seleksjonstest 4

raskest på denne spørringen. Rindex 0.1.2 viser seg å være den raskeste her. Dette kan skyldes at FilmRating relasjonen har mange færre rader. Bruken av iterator framfor direkte traversering av en array koster en del mer tid, og det er sannsynlig at denne forskjellen blir mer signifikant når relasjonen er såpass liten at andre faktorer som gjør Rindex 0.3 raskere på større relasjoner ikke kommer til syne. En annen konsekvens av at FilmRating relasjonen ikke er så stor, er at Rindex 0.1.1 ikke påvirkes så mye av I/O håndteringen at den yter dårligere enn Oracle. Det er likevel en signifikant forbedring i ytelse fra Rindex 0.1.1 til Rindex 0.1.2, som også delvis kan skyldes feilen som gjør at `AND`-ledd blir borte.

Spørring 5

```
SELECT * FROM Participation
WHERE partname <> '1' AND partname <> '2'
  AND partname <> '3' AND partname <> '5';
```

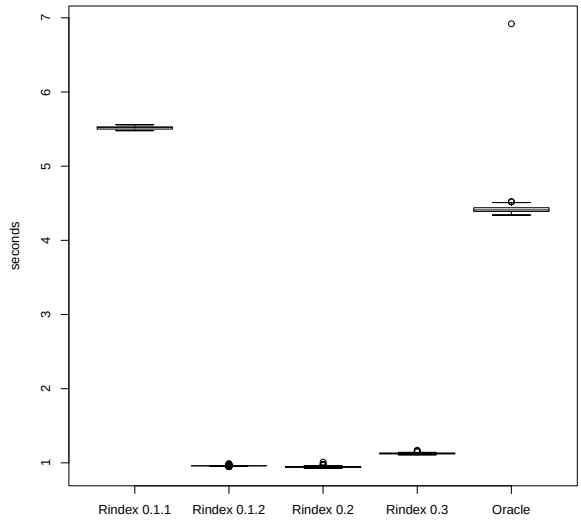
	Rindex 0.1.1	Rindex 0.1.2	Rindex 0.2	Rindex 0.3	Oracle
Min.	5.480	0.9500	0.9300	1.110	4.340
1st Qu.	5.500	0.9600	0.9400	1.120	4.390
Median.	5.520	0.9600	0.9400	1.130	4.410
3rd Qu.	5.530	0.9600	0.9500	1.130	4.440
Max.	5.560	0.9900	1.0100	1.170	6.920
Mean.	5.516	0.9602	0.9453	1.127	4.443

Tabell 8.5: Spørring 5

Dette er en spørring mot en større relasjon, Participation. Resultatene vist i tabell 8.5 viser igjen den store ytelsesforbedringen mellom Rindex 0.1.1 og Rindex 0.1.2. På denne spørringen er Rindex 0.1.2 og Rindex 0.2 omtrent like raske, mens Rindex 0.3 er noe tregere. Nærmere analyse viser at det er seleksjonen som er tregere i Rindex 0.3. Dette forklares igjen med bruken av iterator. Denne spørringen har ingen substitusjoner, som gir Rindex 0.3 en fordel, og resultatsettet er ikke stort nok til at projeksjonen kan gi Rindex 0.3 raskere tid. Denne testen viser dermed helt klart en svakhet i seleksjonen til Rindex 0.3 i forhold til Rindex 0.1.2. Det er vanskelig å finne noen grunn til at Rindex 0.2 er såvidt raskere enn Rindex 0.1.2 på denne spørringen, og det kan skyldes forskjeller i optimaliseringen gjort av kompilatoren.

Spørring 6

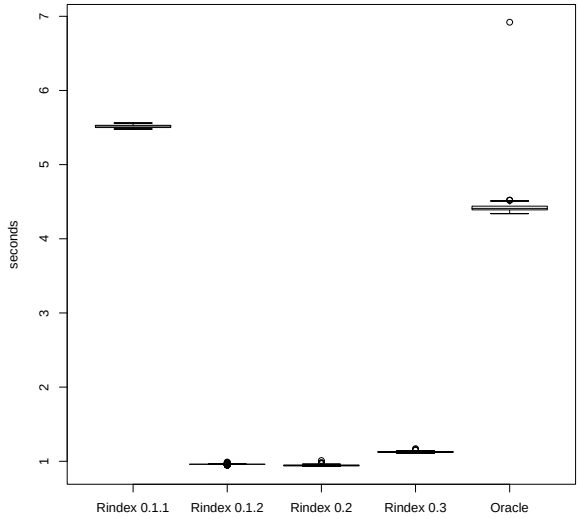
```
SELECT * FROM Participation
WHERE pid <> 1 AND pid <> 2
  AND pid <> 3 AND pid <> 5;
```



Figur 8.5: Seleksjonstest 5

	Rindex 0.1.1	Rindex 0.1.2	Rindex 0.2	Rindex 0.3	Oracle
Min.	5.470	0.9500	0.9400	1.010	4.360
1st Qu.	5.500	0.9600	0.9500	1.020	4.400
Median.	5.510	0.9600	0.9600	1.020	4.440
3rd Qu.	5.530	0.9700	0.9600	1.030	4.462
Max.	5.580	0.9900	1.0100	1.080	7.210
Mean.	5.512	0.9641	0.9584	1.027	4.500

Tabell 8.6: Spørring 6



Figur 8.6: Seleksjonstest 6

Denne testen viser den samme trenden som spørring 5 gjorde, og forklaringene på ytelsesforskjellene er de samme. Vi kan merke oss at Rindex 0.3 er noe raskere på denne spørringen, enn på spørring 5. Det er vanskelig å si nøyaktig hvorfor det er slik, men en mulig teori er at Rindex 0.3 ikke håndterer like verdier så bra. Attributtet *Partname* i Participation-relasjonen har nemlig kun 5 forskjellige verdier fordelt på de 232 598 tuplene. En annen mulig forklaring kan være at det er strenghåndteringen som er for dårlig i Rindex. Hvis denne ytelsesforskjellen mellom sammenligning av tall, og strenger med kun ett tegn skyldes strenghåndteringen, vil dette komme veldig klart fram ved å bruke en test som sammenligner større strenger. Jeg har ikke utført noen slik test, da dette ikke er hensikten med denne oppgaven.

8.2.2 Join

For å teste join er det utformet en join-spørring mot hver database. Spørringen mot nummerdatabasen joiner alle de tre relasjonene, mens spørringen mot filmdatabasen joiner de tre største relasjonene.

Join 1

```
SELECT *
FROM ((X INNER JOIN Y ON X1 = Y1)
      INNER JOIN Z ON X1 = Z1);
```

	Rindex 0.1.1	Rindex 0.1.2	Rindex 0.2	Rindex 0.3	Oracle
Min.	119.6	21.44	21.62	20.93	108.3
1st Qu.	125.0	21.47	21.73	21.01	109.0
Median.	125.5	21.49	21.79	21.04	109.3
3rd Qu.	125.8	21.52	21.88	21.07	109.8
Max.	126.3	21.86	22.16	21.49	112.4
Mean.	124.7	21.51	21.81	21.05	109.4

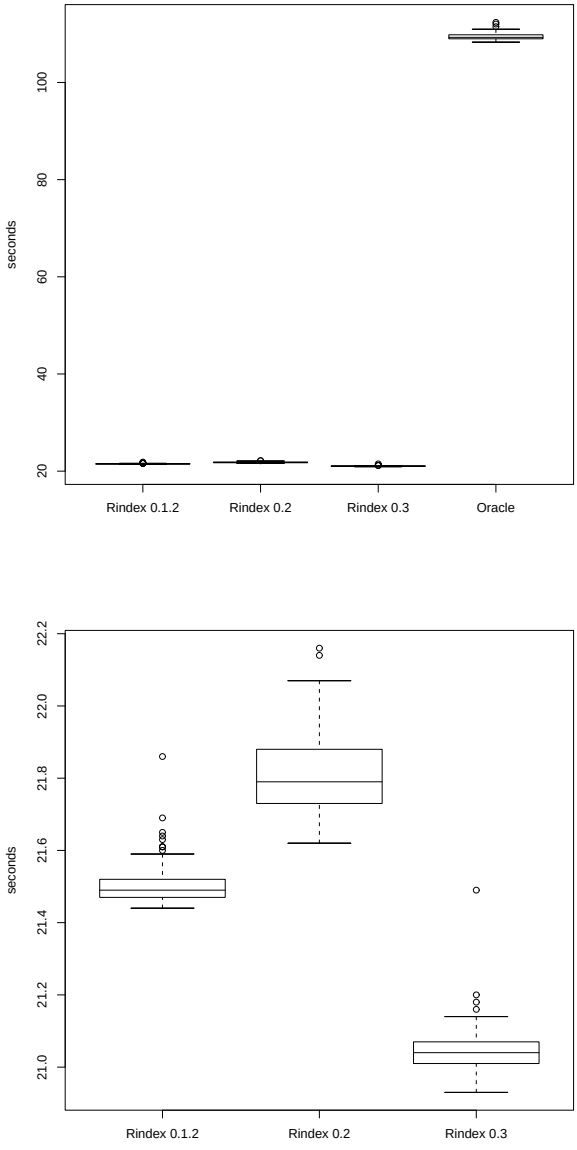
Tabell 8.7: Join 1

5-nummer oppsummeringen viser at forskjellen i ytelse mellom Rindex 0.1.1 og Rindex 0.1.2 er enda større enn for seleksjon, noe som skyldes at join-spørringen returnerer mange flere tupler. Det er derimot mindre forskjeller i ytelse mellom Rindex 0.1.2, Rindex 0.2 og Rindex 0.3. Det er vanskelig å se noen forskjeller mellom implementasjonene i det hele tatt i det øverste boksplottet i figur 8.7 på neste side. Dette er fordi Oracle er så mye tregere at boksplottet til denne gjør at y-aksen strekker seg over et stort område og dermed ikke viser forskjeller på tidelssekund nivå noe godt. Et ekstra boksplott, uten Oracle, er derfor vist nederst i samme figur. Her ser vi mye klarere at Rindex 0.2 er noe tregere enn Rindex 0.1.2, mens Rindex 0.3 er den raskeste av de tre. Forskjellene er likevel som sagt relativt små, sett i forhold til eksekverinstidene på i overkant av 21 sekunder. Nærmere analyse viser at den lille tiden Rindex 0.3 sparer inn i forhold til Rindex 0.1.2 er resultatet av en noe mer effektiv projeksjon. Dette tyder på at det er raskere å gjøre oppslag i den nye og enklere inverterte indeksstrukturen. Analysen avslører også at Rindex 0.3 har en noe tregere join-algoritme (i størrelsesorden 0.2 sekunder, altså rundt 30% av tiden til join-algoritmen til Rindex 0.1.2), noe som sannsynligvis skyldes bruken av iterator. Rindex 0.2 på sin side ser ut til å tape tid i join-algoritmen, noe bruken av vektorer må ta skylden for.

Join 2

```
SELECT *
FROM (Film INNER JOIN
        (Participation INNER JOIN Person
        ON Participation.personid = Person.personid)
     ON Film.filmid = Participation.filmid);
```

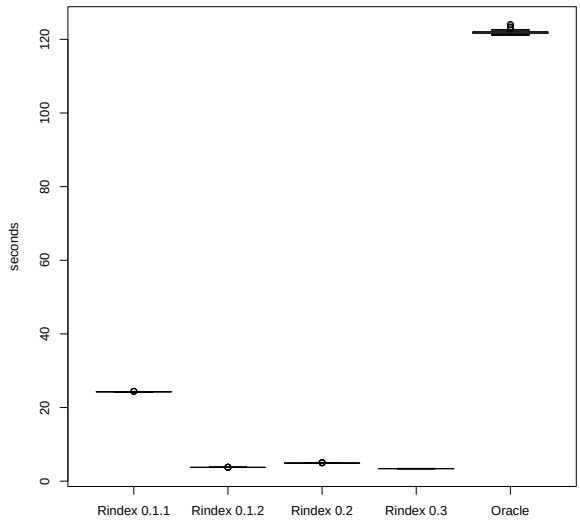
5-nummer oppsummeringen i tabell 8.8 viser den samme trenden som den første join-testen. Rindex 0.3 er noe raskere enn Rindex 0.1.2. Forskjellene kan igjen forklares med raskere projeksjon, i tillegg til at denne spørringen krever en substitusjon, som gir Rindex 0.3 en liten ytelsesforbedring. Analysen av tiden til de forskjellige operasjonene viser imidlertid igjen at Rindex 0.3 er noe tregere enn Rindex 0.1.2 til å utføre selve joinen. På denne spørr-



Figur 8.7: Jointest 1

	Rindex 0.1.1	Rindex 0.1.2	Rindex 0.2	Rindex 0.3	Oracle
Min.	24.19	3.740	4.880	3.380	121.2
1st Qu.	24.25	3.750	4.900	3.390	121.6
Median.	24.28	3.750	4.910	3.410	121.8
3rd Qu.	24.30	3.760	4.922	3.420	122.1
Max.	24.42	3.800	5.070	3.450	124.0
Mean.	24.28	3.756	4.918	3.406	121.9

Tabell 8.8: Join 2



Figur 8.8: Jointest 2

ingen bruker Rindex 0.3 ca 0.3 sekunder på å utføre de to joinoperasjonene, mens Rindex 0.1.2 bruker rundt 0.2 sekunder.

8.2.3 Øvrige spørringer

I tillegg til de foregående spørringene beskriver [12] også resultatene av to andre tester. Den første var en test av projeksjon. Hensikten med denne spørringen var å se om det å kun velge ett attributt påvirker ytelsesforskjellen mellom Rindex og Oracle. Dette er en test som nok er mindre interessant i ytelsestestene som sammenligner de forskjellige Rindex implementasjonene, siden disse alle implementerer projeksjon på samme måte (med unntak av Rindex 0.1.1 som mangler I/O optimaliseringene, og dermed naturlig nok vil få dårligere ytelse). Jeg har likevel valgt å ta denne testen med, men da mest som en ekstra join-test og en test mellom Oracle med og uten Rindex.

Den andre av disse spørringene hadde som hensikt å teste om Rindex kunne dra nytte av å dytte seleksjoner i optimaliseringen. Resultatene av denne testen var at Rindex gjorde det dårligere enn Oracle på denne spørringen. Dette ble da forklart med at Rindex kunne dra stor nytte av å dytte seleksjoner, men den feilen som ble beskrevet i 8.1.4 må ta sin del av skylden for den dårlige ytelsen til Rindex på denne spørringen. Når Rindex returnerer nesten dobbelt så mange tupler som Oracle, og i tillegg benytter en treg utskriftsrutine, er det naturlig at dette påvirker resulatet i stor grad. Denne testen er det derfor interessant å gjøre på nytt, uten disse problemene i Rindex.

Projeksjon

```
SELECT X1
FROM ((X INNER JOIN Y ON X1 = Y1)
      INNER JOIN Z ON X1 = Z1);
```

Hvis vi ser på 5-nummer oppsumeringen til dataene fra projeksjonstesten ser vi tydelig at det å velge kun ett attributt framfor tre attributter medfører en kraftig ytelsesforbedring både for Oracle og for alle Rindex implementasjonene. Noe oppsiktsvekkende med disse resultatene er at Rindex 0.3 ikke er den raskeste av Rindex implementasjonene, siden Rindex 0.3 var raskest på join-spørringen som velger alle attributtene (se tabell 8.7). Det er Rindex 0.1.2 som har best ytelse på denne spørringen. Som for join-spørringen lengre opp, gjør eksekveringstidene til Oracle det vanskelig å se forskjellene mellom Rindex implementasjonene i boksplottet, derfor er det igjen vist to plott i figur 8.9 på side 55. At Rindex 0.3 er tregere enn Rindex 0.1.2 på denne spørringen, skyldes to ting. For det første er join-algoritmen til Rindex 0.3 noe tregere enn den til Rindex 0.1.2, som nevnt tidligere. I tillegg viser det seg at projeksjonen er raskere i Rindex 0.1.2 enn i Rindex 0.3 for denne spørringen. Dette kan skyldes at det her kun er valgt ett attributt,

	Rindex 0.1.1	Rindex 0.1.2	Rindex 0.2	Rindex 0.3	Oracle
Min.	20.21	2.960	4.20	3.440	43.20
1st Qu.	20.61	2.980	4.27	3.470	43.83
Median.	20.74	2.990	4.29	3.480	44.08
3rd Qu.	20.83	3.010	4.32	3.490	44.37
Max.	21.08	3.060	4.53	3.550	45.39
Mean.	20.70	2.995	4.30	3.484	44.12

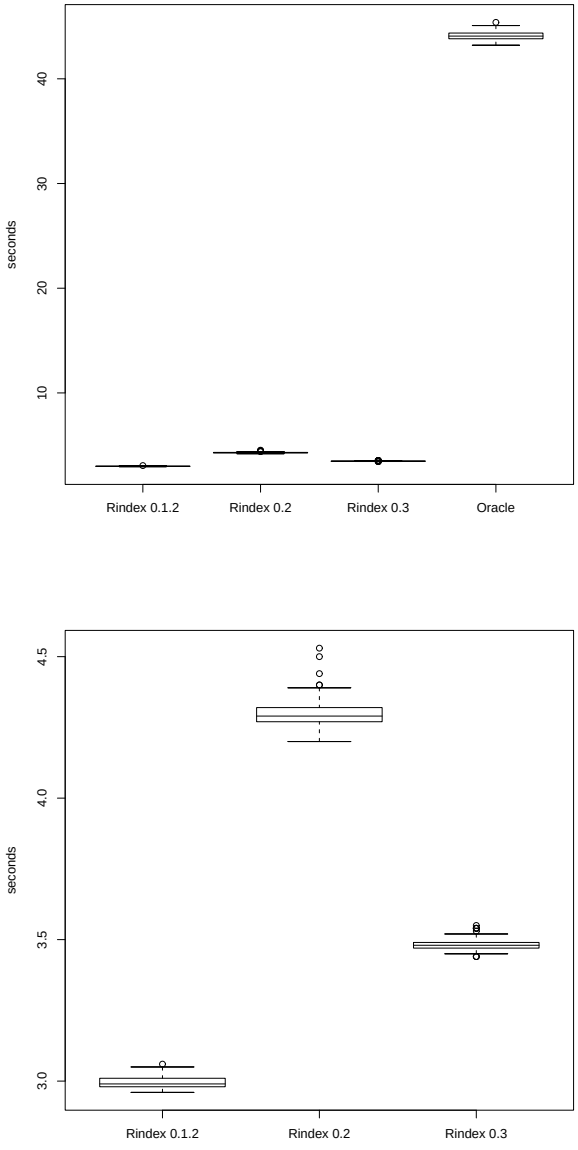
Tabell 8.9: Projeksjon

og at det dermed blir langt færre oppslag i den inverterte datastrukturen. I steden kommer svakheten til iteratoren fram, siden den ekstra tiden dette koster utgjør en større del av den totale eksekveringstiden til projeksjonen.

Join med seleksjon

```
SELECT *
FROM ((X INNER JOIN Y ON X1 = Y1)
        INNER JOIN Z ON X1 = Z1)
WHERE (X2 > 20000 AND X2 <= 80000)
AND (Y2 > 20000 AND Y2 <= 80000)
AND (Z2 > 20000 AND Z2 <= 80000);
```

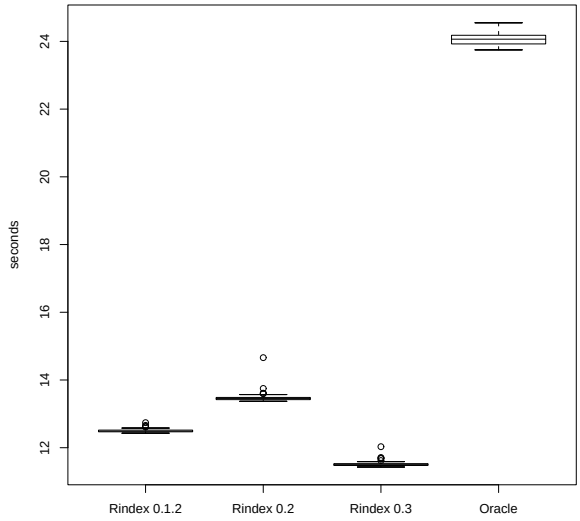
For denne spørringen er resultatene snudd på hodet i forhold til de som er beskrevet i [12]. Vi ser at Rindex 0.1.1 har mye dårligere ytelse enn både de andre Rindex implementasjonene og Oracle. Dette skyldes både den feilen som er beskrevet over, og I/O håndteringen. De øvrige Rindex implementasjonene er rundt dobbelt så raske som Oracle. På denne spørringen er igjen Rindex 0.3 den raskeste, med rundt 1 sekund bedre tider enn Rindex 0.1.2. Rindex 0.2 taper igjen litt til de to andre. Boksplottet i figur 8.10 på side 56 viser forskjellene tydelig. En grundig analyse av hvilke operasjoner som bruker tid, viser at Rindex 0.3 er ca 1 sekund raskere på projeksjonen til denne spørringen (5 sekunder mot 6 sekunder i Rindex 0.1.2). Resten av operasjonene har de samme forskjellene i ytelse som beskrevet tidligere, og det viser seg at det er det sekundet Rindex 0.3 sparer inn i projeksjonen som blir avgjørende for resultatet av denne testen. Dette resultatet viser at Rindex ikke behøver å dytte seleksjoner for å være raskere enn Oracle (selve joinen utgjør ikke mer enn 0.8 sekunder av den totale tiden til Rindex 0.3). Det er likevel ikke mulig å trekke noen slutninger angående om Rindex kan være enda raskere ved å dytte seleksjoner.



Figur 8.9: Projeksjonstest

	Rindex 0.1.1	Rindex 0.1.2	Rindex 0.2	Rindex 0.3	Oracle
Min.	54.66	12.43	13.37	11.43	23.75
1st Qu.	56.65	12.47	13.42	11.48	23.93
Median.	56.84	12.49	13.46	11.50	24.06
3rd Qu.	57.02	12.52	13.48	11.53	24.18
Max.	57.43	12.74	14.66	12.03	24.55
Mean.	56.58	12.50	13.47	11.51	24.06

Tabell 8.10: Join med seleksjon



Figur 8.10: Join med seleksjon test

8.3 Konklusjoner

Disse testene har vist at den nye inverterte indeksstrukturen tillater noe raskere oppslag enn den gamle, og at CSB^+-trær fungerer like godt som datastruktur for indeksene som sekvensielle indekser når det gjelder spørringer. Testene har også vist at iteratoren som brukes for å traversere trærne trenger noen forbedringer for å kunne yte bedre i forhold til å løpe igjennom en array. Union-algoritmen til Rindex 0.3 ser i tillegg ut til å være endel raskere enn union-algoritmen til Rindex 0.1.2. Bruken av vektorer i Rindex 0.2 viser seg å gjøre denne versjonen tregere på nesten samtlige av spørringene som er testet her. Alt i alt kan det se ut til at CSB^+-trær er en god erstatning for sekvensielt sorterte data når oppdateringer også skal støttes, selv om det ennå gjenstår noen mulige optimaliseringer.

Kapittel 9

Ytelsestester av oppdateringer

9.1 Formål og testmetode

For å finne ut om det var hensiktsmessig å reimplementere indeksstrukturen til Rindex for å støtte oppdateringer, eller om den sekvensielle indeksstrukturen var god nok, må ytelsen i utførelsen av oppdateringer sammenlignes mellom de to implementasjonene.

9.1.1 Hva skal testes?

Å teste ytelsen til oppdateringer mellom Rindex 0.2 og Rindex 0.3 bør gjøres uten å sende oppdateringene til disk. Å skulle sende oppdateringer til disk ville bety at resultatet ble påvirket av ytelsen til Oracle. Dette er ikke ønskelig, både fordi det er sannsynlig at dette vil representere en stor del av eksekveringstiden, og fordi Oracle har større spredning i eksekveringstider enn Rindex. Ytelsestestene mellom Rindex 0.2 og Rindex 0.3 vil derfor bli utført uten den delen av koden som sender oppdateringene til disk.

I tillegg til å finne ut hvilken av de to implementasjonene som er raskest, kan det være interessant å teste ytelsen til oppdateringer via Rindex sammenlignet med Oracle uten Rindex. Denne sammenligningen må gjøres med to forskjellige databaser i Oracle, en med og en uten sekundærindekser. Dette fordi Rindex er en annen måte å håndtere sekundærindekser på. Det er rimelig å anta at Rindex vil ha dårligere ytelse enn Oracle uten sekundærindekser, siden oppdateringene også sendes til Oracle fra Rindex. Formålet med disse testene er da å se om Rindex[1] kan utføre oppdateringer raskere enn Oracle med sekundærindekser. Oracle uten sekundærindekser er tatt med som referanse.

Jeg har valgt å ikke teste ytelsen på `CREATE`- og `DROP`-setninger, både fordi disse har insignifikant kost, og fordi dette er oppdateringer som ikke

[1] I dette kapitlet referere Rindex til Oracle med Rindex som overbygg i testene mellom Rindex og Oracle.

gjøres så ofte.

9.1.2 Testmetode

Gjennomføring av testene

Testmetoden er hovedsakelig den samme som den beskrevet i 8.1.3, men hva som måles vil være noe forskjellig. Under test av spørringer er det tiden det tar å utføre en enkelt spørring som er interessant, men tiden det tar å utføre en enkelt `INSERT` f.eks kan være så liten at det er vanskelig å se noen forskjeller. Noen tester vil derfor være en sekvens av oppdateringer, og målingene vil bli utført på hele sekvensen. Testskriptet som benyttes er det samme, men SQL-filene som sendes til Rindex og Oracle vil inneholde sekvenser av oppdateringer, og tiden som måles er da tiden det tar å eksekvere alle oppdateringene i denne fila. Det vil også bli kjørt noen tester der målingene utføres på kun en oppdaterings-setning.

Fordi oppdateringer endrer innholdet i databasen, er det nødvendig å resette innholdet i databasen mellom hver test, slik at alle testene blir utført med like forhold. Dette gjøres ved å starte alle testene med en `CREATE`-setning, og avslutte dem med en `DROP`. For testingen av `UPDATE` og `DELETE` må også all data settes inn i databasen på nytt for hver test, med `INSERT`-setninger.

Presentering av resultatene

Resultatene av disse testene vil bli presentert med gjennomsnittsverdi. 5-nummeroppsummering vil ikke bli brukt her, for det ville blitt en uoversiktlig mengde med data. I tillegg vil de gjennomsnittlige eksekveringstidene plottes med hensyn på antall tupler i relasjonen testene er utført på. Ikke alle plottene er vist her.

9.1.3 Testdata

Alle testene vil bli utført på relasjonen X fra nummerdatabasen. Dette er den eneste relasjonen som vil være i databasen under testing. Oppdateringssetningene vil bli generert ut fra dataene til X relasjonen, og dette gir mulighet til å sette inn og endre på data i tilfeldig rekkefølge på en enkel måte.

9.2 Testene

9.2.1 Innsetting

Testene

`INSERT`-setningene blir generert av et skript som bruker dataene til X relasjonen og lager n antall `INSERT`-setninger, der n er parameter til skriptet, og

skriver disse til en fil. Hvis setningene er generert for testing av Rindex mot Oracle skrives `COMMIT` på slutten av fila. Dette skriptet er brukt til å generere filer med 100, 1000, 5000, 10 000 og 100 000 `INSERT`-setninger. Dette gjør det mulig å se hvor mange tupler som må til før det er noen signifikante ytelsesforskjeller å se. Aller helst skulle det vært flere steg med forskjellige antall mellom 5000 og 100 000, men fordi det tar ganske lang tid å kjøre 100 tester av en sekvens med `INSERT`-setninger, har jeg måttet avstå fra å ta med flere steg. I testene utført med database backend er kun stegene med 1000, 10 000 og 100 000 tatt med, fordi det tar enda lengre tid å kjøre slike tester med database backend. Disse stegene i antall tupler er brukt for `UPDATE`- og `DELETE`-tester også.

Resultater

	Rindex 0.2	Rindex 0.3
100	0.0705	0.07
1000	0.1315	0.1196
5000	2.765	0.3326
10000	12.11	0.6081
100000	1378	5.947

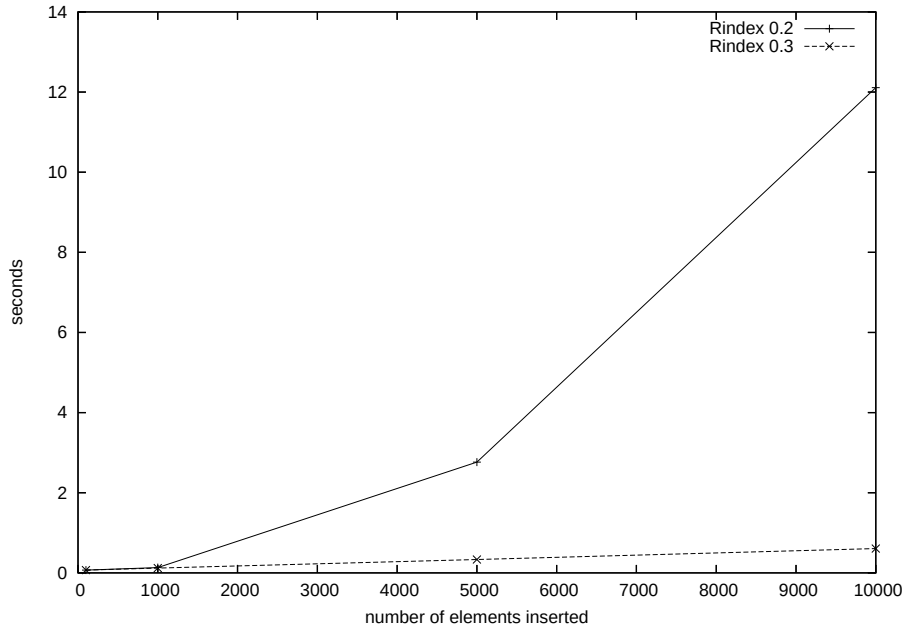
Tabell 9.1: Innsetting uten skriving til disk

Uten database backend Hvis vi ser på gjennomsnittstidene for innsetting til Rindex 0.2 og Rindex 0.3, vist i tabell 9.1, så ser vi at Rindex 0.2 allerede ved 5000 innsettinger er mange ganger tregere enn Rindex 0.3. Rindex 0.3 klarer å gjøre 10 000 innsettinger på litt over et halvt sekund, mens Rindex 0.2 bruker 12.11 sekunder på det samme antallet. For 100 000 innsettinger tar det så lang tid som 1378 sekunder, eller litt over 20 minutter for Rindex 0.2. Dette viser at Rindex 0.2 ikke takler mange innsettinger etter hverandre. Rindex 0.3 derimot klarer å gjøre 100 000 innsetting på ca. 6 sekunder. Grunnen til at Rindex 0.2 bruker så lang tid på innsettinger er at det tar lineær tid å sette inn et nytt element i en sekvensielt sortert liste. Worst-case for n innsettinger er da $O(n * n)$.

Plottet i figur 9.1 viser trenden i tidsøkning i forhold til antall insert. Testen med 100 000 innsettinger er utelatt fra dette plottet.

	Rindex 0.3	Oracle uten indekser	Oracle med indekser
1000	1.003	1.333	1.474
10000	9.701	11.15	12.17
100000	100.50	115.3	122.4

Tabell 9.2: Innsetting med skriving til disk



Figur 9.1: Insert-tester uten database backend

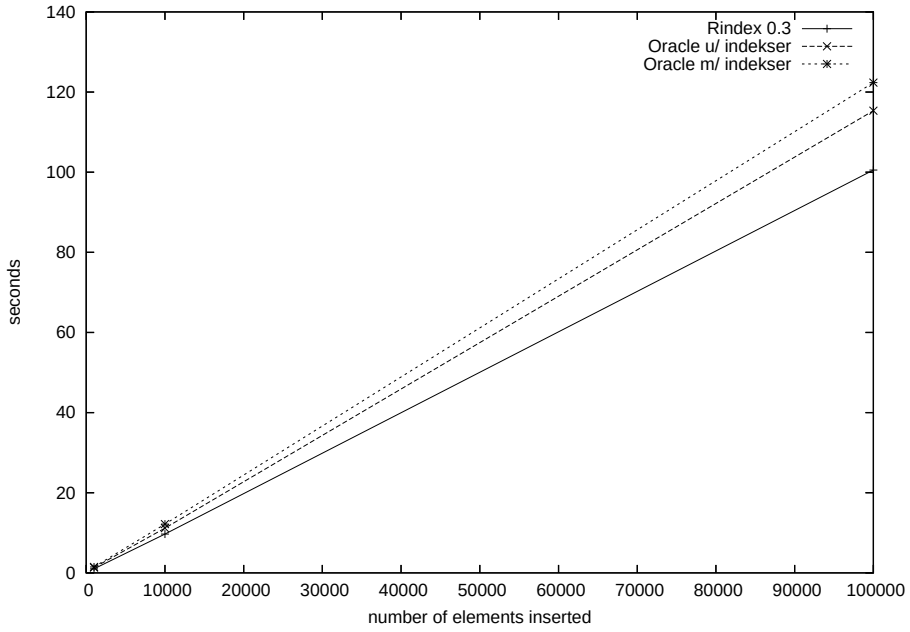
Med database backend Hvis vi ser på gjennomsnittsverdiene fra sammenligning mellom Rindex 0.3 og Oracle, vist i tabell 9.2, så ser vi at Rindex er noe raskere enn både Oracle uten sekundærindekser på attributtene, og Oracle med sekundærindekser på attributtene. Figur 9.2 viser trenden i tidsutvikling i forhold til antall innsettinger. Årsaken til at Rindex er raskest på denne testen, selv om endringene i Rindex også sendes til Oracle, kan være at kommunikasjonen mellom C++ og Oracle via sqlapi'et er raskere enn kommunikasjonen mellom sqlplus og Oracle.

9.2.2 Oppdateringer

Testene

Testing av `UPDATE`-setninger har to forskjellige tilnærminger. Den ene er å teste en sekvens av `UPDATE`-setninger som kun oppdaterer ett tuppel hver, slik som for `INSERT`. Den andre er å teste en enkelt `UPDATE`-setning som oppdaterer et stort antall tupler. Innenfor denne siste typer er det også to typer. En `UPDATE`-setning kan inneholde et omfattende `WHERE`-ledd som skaper endel prosessering, uten å fjerne så mange tupler, og en `UPDATE`-setning kan også være uten et `WHERE`-ledd, og dermed oppdatere alle tupler i databasen. Jeg har valgt å teste alle disse tre variantene, siden de tester forskjellige aspekter ved oppdateringer.

Den første testen blir generert på samme måte som for `INSERT`, der



Figur 9.2: Insert-tester med skriving til disk

`UPDATE`-setingene lages av et skript som bruker dataene til X-relasjonen, og skriver n setninger til en fil. Jeg har valgt å sette n til 100, og kjøre denne sekvensen av 100 `UPDATE`-setninger på X-relasjonen med forskjellig antall tupler. Stegene i antall tupler er de samme som for `INSERT`-testene. Dette gjør det mulig å se hvordan størrelsen på relasjonen påvirker ytelsen til oppdateringer. Setningene er alle på formen:

```
UPDATE X SET
  x2 = b,
  x3 = c
WHERE x1 = a;
```

b og c blir satt av skriptet til å være én mer enn den opprinnelige verdien til henholdsvis x2 og x3 i tuppelet. Å spørre på primærnøkkelen x1 sørger for raskt oppslag i tillegg til garanti om at kun ett tuppel blir oppdatert.

De øvrige testene er som følger:

1. ```
   UPDATE X SET
     X2 = 100,
     X3 = 200
   WHERE (X1 < 49000 OR X1 > 51000)
      OR (X2 < 49000 OR X2 > 51000)
      OR (X3 < 49000 OR X3 > 51000);
   ```
   ```

```
2. UPDATE X SET
     X2 = 100,
     X3 = 200
   WHERE (X1 <> 1 AND X1 <> 2)
     AND (X2 <> 3 AND X2 <> 5)
     AND (X3 <> 8 AND X3 <> 13);

3. UPDATE X SET
     X2 = 100,
     X3 = 200;
```

De to første av disse testene er utformet for å være så like som mulig spørring 2 og 3 fra avsnitt 8.2.1. De skal kreve endel prosessering samtidig som nesten alle tupler blir oppdatert. Den første testen oppdaterer ikke alle tuplene, mens den andre testen oppdaterer alle tuplene.

Den tredje og siste testen skal oppdatere alle tuplene i relasjonen. En bieffekt av disse tre testene er at det blir veldig mange like verdier i databasen. Ved å kjøre andre spørringer etter disse testene kan det avdekkes svakheter i håndteringen av like verdier.

Resultater

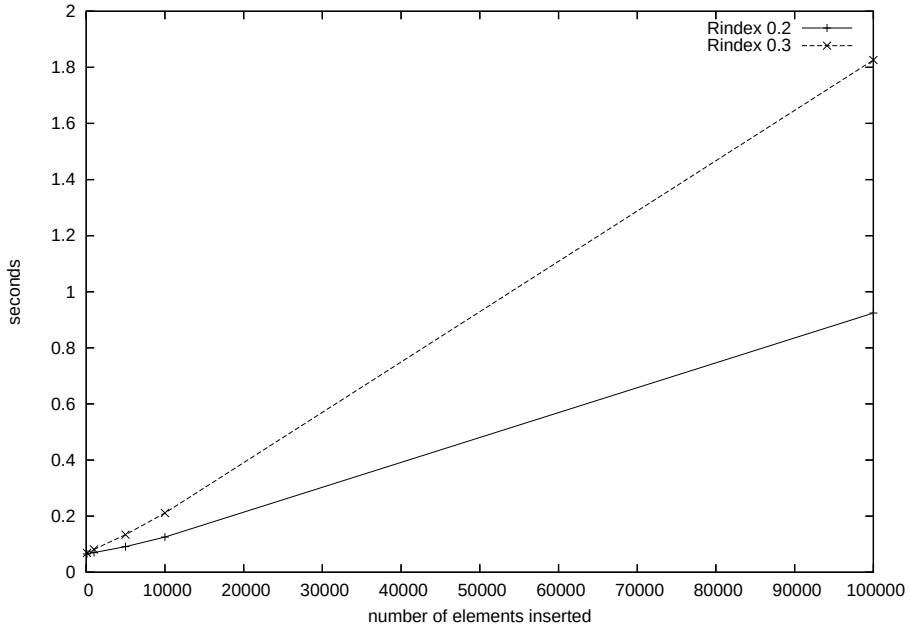
	Rindex 0.2	Rindex 0.3
100	0.0705	0.0717
1000	0.0876	0.072
5000	0.2818	0.0729
10000	0.675	0.0725
100000	13.16	0.0783

Tabell 9.3: Update-sekvens uten database backend

Update-sekvens uten database backend Hvis vi ser på resultatene i tabell 9.3, så ser vi at Rindex 0.3 er nesten like rask til å oppdatere 100 av 100 000 tupler, som 100 av 100 tupler. Rindex 0.2 derimot bruker omtrent like mye tid som Rindex 0.3 på å oppdatere 100 av 100 tupler, men bruker så mye som rundt 13.16 sekunder på å oppdatere 100 av 100 000 tupler. Dette er 168 ganger mer enn Rindex 0.3. Grunnen til at Rindex 0.2 er så treg til å utføre disse oppdateringene, er at for hver oppdaterings-setning blir de gamle verdiene byttet ut med den nye, på samme plass i minnet. Deretter blir dataene sortert på nytt med quicksort. En slik serie med oppdateringer medfører derfor 100 sorteringer med quicksort, på nesten ferdig sortert input, siden kun ett element er endret.

	Rindex 0.2	Rindex 0.3
100	0.065	0.0689
1000	0.0701	0.0811
5000	0.0912	0.134
10000	0.1252	0.2111
100000	0.9243	1.826

Tabell 9.4: Update-test 1 uten database backend



Figur 9.3: Update-test 1 uten database backend

Update-test 1 uten database backend Tabell 9.4 viser en helt annen trend i resultatene fra denne testen i forhold til den forrige. Denne gangen er det nesten like mange tupler som det er i relasjonen som blir oppdatert, men alle oppdateringene er resultatet av samme oppdateringssetning. Rindex 0.2 har mye bedre ytelse på en slik oppdatering. Dette er fordi dataene kun blir sortert én gang, etter at alle verdiene er oppdatert. Rindex 0.3 derimot utfører en serie med delete-operasjoner på CSB^+-treet, etterfulgt av en serie med insert-operasjoner. Dette er operasjoner som i seg selv er meget raske, uansett hvor stor relasjonen er, men hvis det er mange tupler som skal oppdateres, så tar summen av alle disse operasjonene endel tid. Dataene er plottet i figur 9.3.

	Rindex 0.2	Rindex 0.3
100	0.0634	0.0661
1000	0.0703	0.0777
5000	0.0903	0.1303
10000	0.1182	0.2004
100000	0.8416	1.63

Tabell 9.5: Update-test 2 uten database backend

Update-test 2 uten database backend Tabell 9.5 viser små forskjeller i tider i forhold til den forrige testen. Dette er som forventet. Grunnen til at disse tidene er noe lavere enn tidene vist i tabell 9.4 er at denne testen ikke krever noen union-operatorer. Det er altså eksekveringen av `WHERE`-leddet som er raskere her.

	Rindex 0.2	Rindex 0.3
100	0.0631	0.0674
1000	0.0646	0.0716
5000	0.0704	0.1126
10000	0.0725	0.17
100000	0.1806	1.262

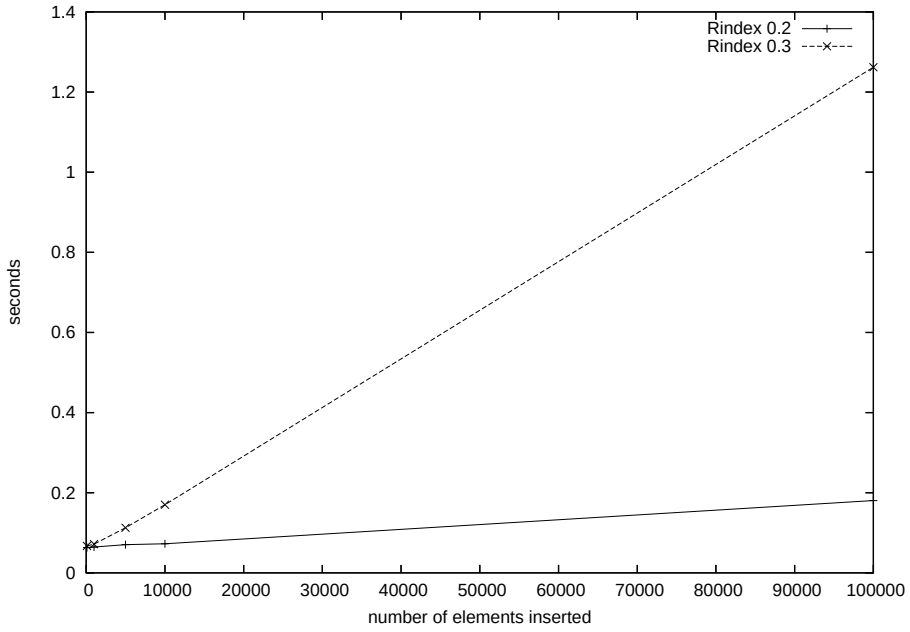
Tabell 9.6: Update-test 3 uten database backend

Update-test 3 uten database backend Den fjerde og siste oppdateringstesten viser igjen den samme trenden i tider. Tidene vist i tabell 9.6 er som forventet enda noe lavere enn for de to foregående testene, siden denne oppdaterings-setningen ikke har noe `WHERE`-ledd i det hele tatt. Disse tidene viser derfor kun tiden for å utføre selve oppdateringene på indeksene.

	Rindex 0.3	Oracle uten indekser	Oracle med indekser
1000	0.1643	0.2572	0.29
10000	0.1635	0.3013	0.3112
100000	0.2062	0.3595	0.3795

Tabell 9.7: Update-sekvens med database backend

Update-sekvens med database backend Resultatene i tabell 9.7 viser, slik som for insert, at en sekvens med update-setninger som oppdaterer ett tuppel hver, er raskere å utføre i Rindex enn i Oracle. Dette tyder igjen på at det er raskere å kommunisere med Oracle via sqlapi'et som brukes av C++ enn med sqlplus. Figur 9.5 viser et plott av disse resultatene. Vi legger merke til i plottet at gjennomsnittstiden for eksekvering av 100 oppdateringer i en



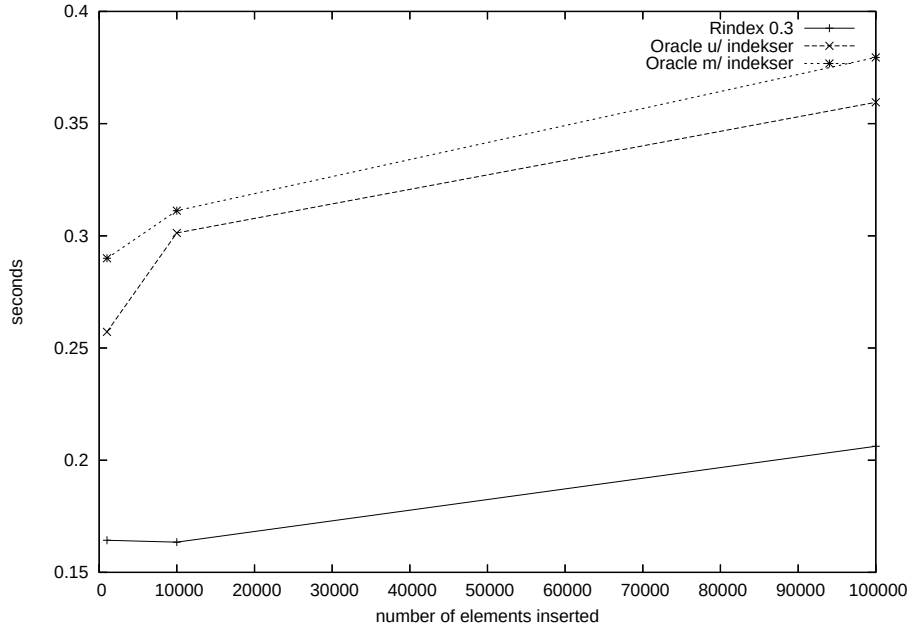
Figur 9.4: Update-test 3 uten database backend

relasjon med 10 000 tupler er raskere enn for 1000 tupler i Rindex. Dette er sannsynligvis tilfeldig, og viser egentlig bare at det er like raskt å gjøre 100 oppdateringer blant 10 000 tupler som blant 1000 tupler.

	Rindex 0.3	Oracle uten indekser	Oracle med indekser
1000	0.1165	0.1935	0.2611
10000	0.4509	0.3225	1.494
100000	4.805	2.960	879.80

Tabell 9.8: Update-test 1 med database backend

Update-test 1 med database backend Hvis vi ser på resultatene av denne testen, vist i tabell 9.8, så ser vi at Rindex er raskest i eksekveringen av oppdateringen for 1000 tupler, men for 10 000 tupler er Oracle uten sekundærindekser raskest. Dette er den første testen som viser Rindex' styrke i forhold til sekundærindekser i Oracle. På 100 000 tupler bruker Oracle rundt 880 sekunder, altså nesten 15 minutter, på å utføre oppdateringen. Rindex bruker kun ca. 4.8 sekunder, som ganske nøyaktig tilsvarer tiden vist i tabell 9.4 for Rindex 0.3 uten database backend, pluss de 2.96 sekundene Oracle uten sekundærindekser bruker.



Figur 9.5: Update-sekvens med database backend

	Rindex 0.3	Oracle uten indekser	Oracle med indekser
1000	0.0914	0.1704	0.2421
10000	0.4006	0.5130	1.330
100000	4.884	2.534	16.03

Tabell 9.9: Update-test 2 med database backend

Update-test 2 med database backend Resultatene vist i tabell 9.9 viser den samme trenden som den forrige testen, med unntak av at Oracle med sekundærindekser kun bruker ca. 16 sekunder på denne oppdateringen, mot de 15 minuttene for forrige test. Dette kan skyldes at den forrige oppdateringssetningen ikke oppdaterer absolutt alle tuplene, noe denne gjør. Det indikerer at Oracle kanskje dropper indeksen før oppdateringene utføres, for så å bygge den på nytt, hvis alle tuplene oppdateres. Dette er i såfall vesentlig raskere enn å måtte oppdatere indeksen direkte. Rindex har ikke dette problemet, og er like effektiv på begge disse testene.

	Rindex 0.3	Oracle uten indekser	Oracle med indekser
1000	0.0915	0.1805	0.2549
10000	0.3522	0.3475	1.191
100000	3.781	2.633	14.89

Tabell 9.10: Update-test 3 med database backend

Update-test 3 med database backend Denne oppdateringen utføres på alle tupler, uten å utføre noen seleksjoner først. Resultatene vist i tabell 9.10 har lignende resultater som de sett i tabell 9.9. Tidene er noe raskere grunnet mangelen på en seleksjon, men bekrefter resultatene som viser at det er mye raskere å ha indekser i Rindex, enn i Oracle.

9.2.3 Sletting

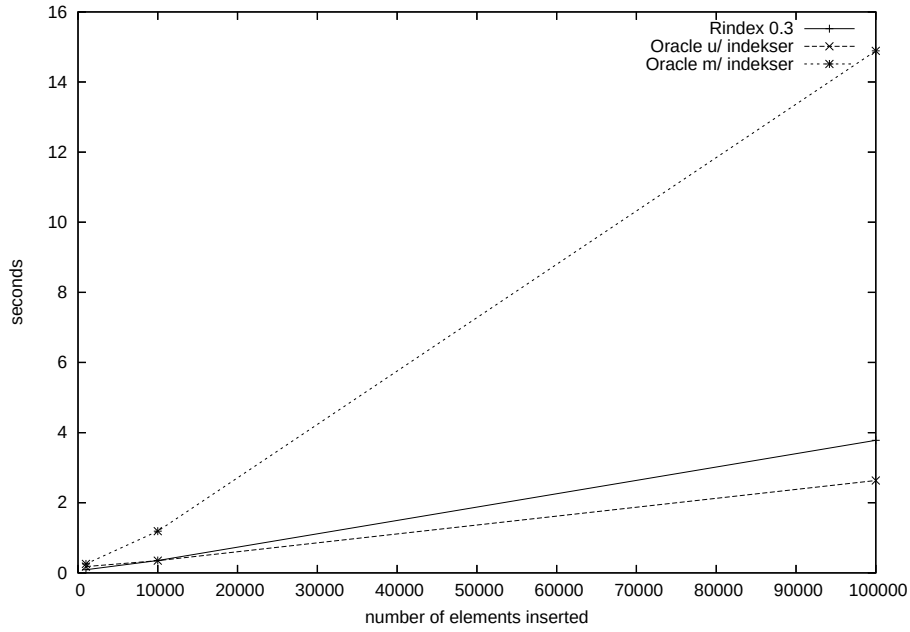
Når det gjelder `DELETE`-setninger, blir utformingen av testene den samme som for `UPDATE`. De samme aspektene kan testes for sletting som for oppdateringer. Den første testen blir da en sekvens av `DELETE`-setninger. Fordi det er raskere å utføre `DELETE` enn `UPDATE` har jeg valgt å la det være 1000 `DELETE`-setninger. Som for `UPDATE` kjøres testene på X-relasjonen med 1000, 10 000 og 100 000 tupler. Setningene er på formen:

```
DELETE FROM X
WHERE x1 = a;
```

De øvrige testene er de samme som for `UPDATE`, men da altså med `DELETE` i steden for:

1. ```
   DELETE FROM X
   WHERE (X1 < 49000 OR X1 > 51000)
      OR (X2 < 49000 OR X2 > 51000)
      OR (X3 < 49000 OR X3 > 51000);
   ```

2. ```
   DELETE FROM X
   WHERE (X1 <> 1 AND X1 <> 2)
   ```



Figur 9.6: Update-test 3 med database backend

```
AND (X2 <> 3 AND X2 <> 5)
AND (X3 <> 8 AND X3 <> 13);
```

3. `DELETE FROM X;`

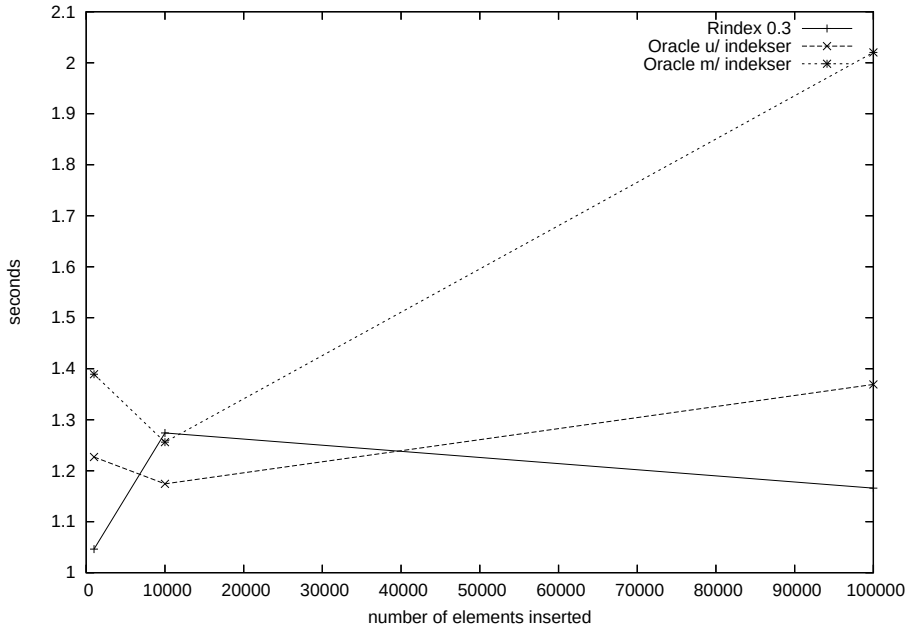
Den siste `DELETE`-setningen har kun som hensikt å teste hvor lang tid det tar å tømme relasjonen for tupler.

Resultater

	Rindex 0.3	Oracle uten indekser	Oracle med indekser
1000	1.046	1.227	1.389
10000	1.274	1.174	1.256
100000	1.166	1.369	2.020

Tabell 9.11: Delete-sekvens med database backend

Delete-sekvens med database backend Resultatene presentert i tabell 9.11 viser at også for en sekvens `DELETE`-setninger er det raskere å utføre dem i Rindex enn i Oracle, i hvertfall hvis det er 100 000 tupler i relasjonen.



Figur 9.7: Delete-sekvens med database backend

	Rindex 0.3	Oracle uten indekser	Oracle med indekser
1000	0.1103	0.1823	0.2237
10000	0.6545	0.5473	1.085
100000	8.086	5.846	491.4

Tabell 9.12: Delete-test1 med database backend

Delete-test 1 med database backend Resultatene av denne testen, presentert i tabell 9.12, viser at det er for delete-setninger som for update-setninger. Rindex bruker noe mer tid enn Oracle uten sekundærindekser grunnet tiden det tar å slette indeksene internt i Rindex, mens Oracle med sekundærindekser har vesentlig dårligere ytelse. Vi ser at for 100 000 tupler bruker Oracle med sekundærindekser ca. 490 sekunder, eller ca. 8 minutter, på å slette nesten alle tuplene i relasjonen.

	Rindex 0.3	Oracle uten indekser	Oracle med indekser
1000	0.1038	0.2176	0.2039
10000	0.6580	0.6339	1.0460
100000	7.638	5.345	11.35

Tabell 9.13: Delete-test 2 med database backend

Delete-test 2 med database backend Denne testen viser igjen at Rindex' måte å håndtere indekser på er raskere enn Oracle's. Vi ser igjen av resultatene i tabell 9.13 at for 100 000 er Oracle med sekundærindekser mye raskere når alle tuplene som slettes, enn når noen skal være igjen. Dette er den samme tendensen som for update-setninger. Vi ser også at for denne testen er det mindre forskjell mellom Rindex og Oracle med sekundærindekser enn det var for oppdateringer. Det er likevel en signifikant forskjell også her.

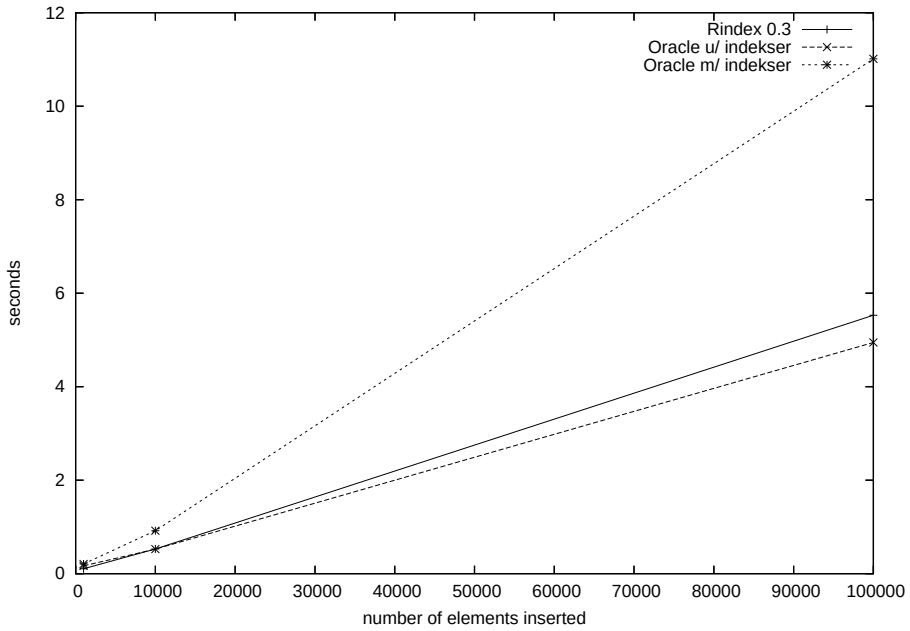
	Rindex 0.3	Oracle uten indekser	Oracle med indekser
1000	0.1048	0.1665	0.2082
10000	0.5294	0.5253	0.9202
100000	5.527	4.946	11.010

Tabell 9.14: Delete-test 3 med database backend

Delete-test 3 med database backend Den siste delete-testen viser klart at det er en god del ekstraarbeid for Oracle å rydde opp i en indeks i tillegg til selve dataene når alle tuplene skal slettes. Resultatene er vist i tabell 9.14, og viser igjen at Rindex er vesentlig raskere enn Oracle med sekundærindekser.

9.3 Konklusjoner

Disse testene har vist at det er mye å hente på å håndtere indekser slik Rindex gjør framfor å ha indekser i Oracle, når det gjelder oppdateringer. Oracle med indekser i Rindex yter betydelig bedre enn Oracle med egne sekundærindekser på samtlige av testene.



Figur 9.8: Delete-test 3 med database backend

Kapittel 10

Avslutning

I innledningen til denne oppgaven ble det satt som mål å utvide Rindex til å støtte oppdateringer på en slik måte at Rindex' håndtering av indekser gir bedre ytelse enn å ha indekser lagret på disk. Oppdateringer ble implementert på to forskjellige indeksstrukturer for å forsøke å nå dette målet. Den første implementasjonen, kalt Rindex 0.2, baserer seg på en sekvensiell indeksstruktur. Rindex 0.3 er den andre implementasjonen, og denne gjør bruk av CSB^+-trær i indekstrukturen. Kapittel 8 viste at både Oracle med Rindex 0.2 og Oracle med Rindex 0.3 yter vesentlig bedre enn Oracle uten Rindex på de spørringene som er testet. I kapittel 9 så vi at Rindex 0.3 viste seg å være en mye mer effektiv implementasjon enn Rindex 0.2 når det gjelder oppdateringer. Vi så også at Rindex 0.3 ga Oracle en signifikant bedre ytelse, også når det gjelder oppdateringer. Dette viser at å lagre indekser i RAM er en god måte å øke ytelsen til tradisjonelle DBMS'er på. Oracle, og andre DBMS'er av samme type, vil få økt ytelse ved å implementere indekser på en lignende måte som Rindex. Ved å implementere dette i DBMS-kjernen kan de åpenbart oppnå en enda bedre ytelsesforbedring enn det Rindex kan gi.

10.1 Videre arbeid

Det er fremdeles mange oppgaver å ta tak i i utviklingen av Rindex. Dette gjelder både hva Rindex støtter, sett utenfra, og hvordan enkelte ting er håndtert internt i Rindex.

En svakhet i Rindex som er verdt å nevne, er håndteringen av like verdier i indeksene. Det er ikke utført noen tester som påviser denne svakheten, men det vil enkelt kunne vises ved å utføre tester på en relasjon med et stort antall tupler, der ett av attributtene kun har to mulige verdier. Svakheten består i at indeksene kun er sortert på nøkkel, og ikke på primærnøkkel innenfor like nøkler. Dette gjør at søk etter like tupler, slik som i union-algoritmen og join-algoritmen, medfører lineære søk. Hvis det er mange like verdier i

en indeks, vil det bli mange lange lineære søk, og dette er ikke effektivt. Oppdaterings- og slettealgoritmen lider også under denne svakheten, siden det her søkes etter gitte tupler. En mulig løsning på dette i Rindex 0.3, som bruker CSB^+-trær, er å ha en peker til et nytt tre, sortert på primærnøkkel, for alle like verdier i treet, i steden for slik det nå gjøres, der like verdier ligger etter hverandre i treet.

Jeg ønsker også å trekke fram noen punkter angående hva Rindex bør støtte, men ikke støtter i dag.

- Rindex mangler støtte for relasjoner med attributter som ikke blir indeksert. Dette gjelder i hovedsak binærdata og store tekster.
- Rindex bør støtte et større subsett av SQL for å kunne uttrykke mer interessante spørringer.
- librindex bør ikke skrive resultatet direkte til stdout, men tilby en kommunikasjonsmetode som tillater å hente ut resultatsettet.
- Kommunikasjonen mellom klient og Rindex-server bør gjøres med sockets i steden for med pipe-filer.
- Rindex støtter ikke flere samtidige brukere, eller kommunikasjon over nettverk.

I tillegg til dette kan det være interessant å se på om det kan være hensiktsmessig å genere indeksene "on-demand", i steden for å genere alle ved oppstart. Dette vil gjøre at Rindex kan startes opp mye raskere, men vil medføre økte kostnader for de første spørringene utført mot hver relasjon.

Tillegg A

Grammatikk

Her følger en beskrivelse av Rindex' utvidete grammatikk (den opprinnelige er inkludert).[1]

<sql>	::=	<select-stmt> | <create> | <drop> | <insert> | <update> | <delete>
<create>	::=	`CREATE TABLE` *relation-name* `(` <create-def> `)`
<create-def>	::=	<attr-def> <attr-def-list> | ϵ
<attr-def-list>	::=	`,` <attr-def> <attr-def-list> | ϵ
<attr-def>	::=	*attribute-name attribute-type* <primary-key>
<primary-key>	::=	`PRIMARY KEY` | ϵ
<drop>	::=	`DROP TABLE` *relation-name*
<insert>	::=	`INSERT INTO` *relation-name* <col-list> `VALUES (` <value-list> `)`
<col-list>	::=	`(` <col-name-list> `)` | ϵ

[1] Terminerende symboler er notert i skrivemaskinskrift, produksjonsymboler er notert <...> og navn er notert i kursiv.

<col-name-list> ::= *attribute-name*
| *attribute-name* `,` <col-name-list>

<value-list> ::= *token*
| *token* `,` <value-list>

<update> ::= `UPDATE` *relation-name* `SET` <update-list> <where>

<update-list> ::= <update-stmt>
| <update-stmt> `,` <update-list>

<update-stmt> ::= *attribute-name* `=` *literal*

<delete> ::= `DELETE FROM` *relation-name* <where>

<select-stmt> ::= <select> <from> <where> <order-by>

<select> ::= `SELECT` <attribute-list>

<attribute-list> ::= <attribute>, <attribute-list>
| <attribute>
| `*`

<attribute> ::= <attribute-name>
| *relation-name*`.*`

<attribute-name> ::= *relation-name*`.`*attribute-name*
| *attribute-name*

<from> ::= `FROM` <join-relation>

<join-relation> ::= `(` <join-relation> `INNER JOIN` <join-relation>
`ON` <join-condition> `)`
| *relation-name*

<join-condition> ::= <attribute-name> `=` <attribute-name>

<where> ::= `WHERE` <condition>
| ϵ

<order-by> ::= `ORDER BY` <attribute-name>
| ϵ

<condition> ::= <boolean-term>
| <condition> `OR` <boolean-term>

<boolean-term> ::= <boolean-factor>
| <boolean-term> `AND` <boolean-factor>

<boolean-factor> ::= <boolean-pri>
| `NOT` <boolean-pri>

<boolean-pri> ::= <comparison>
| `(` <condition> `)`

<comparison> ::= <scalar-exp> <comp-opr> <scalar-exp>
| <attribute-name> <comp-opr> <scalar-exp>

<scalar-exp> ::= <term>
| <sign> <term>
| <scalar-exp> <add-opr> <term>

<term> ::= <factor>
| <term> <mult-opr> <factor>

<factor> ::= *token*
| `(` <scalar-exp> `)`

<comp-opr> ::= `=` | `<>` | `<` | `>` | `<=` | `>=`

<mult-opr> ::= `*` | `/`

<add-opr> ::= `+` | `-`

<sign> ::= <add-opr>

Bibliografi

[1] Englander, Irv. *The Architecture of Computer Hardware and Systems Software. An Information Technology Approach. Second edition.* John Wiley and Sons 2000.

[2] Free Software Foundation. *Bison 1.35.* Internett: `http://www.gnu.org/software/bison/manual/`.

[3] Haavet, Tomas Are: *Rindex, RAM-basert indekshåndtering - en effektivitetsstudie.* Master's thesis, Department of Informatics, University of Oslo, 2005.

[4] Internet Movie Database (IMDB). *http://www.imdb.org*

[5] Lischner, Ray. *STL Pocket Reference.* O'Reilly 2004.

[6] Moore, David S. and McCabe *Introduction to the Practice of Statistics* W. H. Freeman, 2003.

[7] Oracle Online Documentation. `http://www.oracle.com/technology/documentation/index.html`.

[8] Paxson, Vern. *Flex, A fast scanner generator.* Free Software Foundation. Internett: `http://www.gnu.org/software/flex/manual`

[9] Rafienko, Igor V. *Filmdatabasen.* Department of Informatics, University of Oslo, 2002.

[10] Rao, Jun and Ross, Kenneth A. *Making B+- trees cache conscious in main memory. In SIGMOD 00: Proceedings of the 2000 ACM SIGMOD international conference on Management of data,* pages 475-486. ACM Press, 2000.

[11] Tannenbaum, Andrew S. *Modern Operating Systems. Second edition.* Prentice-Hall 2001.

[12] Øierud, Kjell Magne: *Rindex, Ytelsestest av plattform for indekshåndtering i RAM* Master's thesis, Department of Informatics, University of Oslo, 2005.